NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

C2 AT THE EDGE: OPERATING IN A DISCONNECTED LOW-BANDWIDTH ENVIRONMENT

by

Alexander J. Beachy

June 2015

Thesis Co-Advisors:	Gurminder Singh
	John Gibson

REPORT DOCUMENTATION PAGE			*Form Approved OMB No. 0704–0188*
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202–4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.			
1. AGENCY USE ONLY *(Leave Blank)*	2. REPORT DATE 06-19-2015	3. REPORT TYPE AND DATES COVERED Master's Thesis 07-06-2013 to 06-19-2015	
4. TITLE AND SUBTITLE C2 AT THE EDGE: OPERATING IN A DISCONNECTED LOW-BANDWIDTH ENVIRONMENT			5. FUNDING NUMBERS
6. AUTHOR(S) Alexander J. Beachy			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943			8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A			10. SPONSORING / MONITORING AGENCY REPORT NUMBER
11. SUPPLEMENTARY NOTES The views expressed in this document are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. IRB Protocol Number: N/A.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited			12b. DISTRIBUTION CODE

13. ABSTRACT *(maximum 200 words)*

A practical solution to providing effective command, control, and communications (C3) information from the warfighting commander to the edge of the battle space involves using light-weight, handheld devices connected on wireless networks; however, infrastructure-based wireless networks, especially at the edge, are characterized by wildly fluctuating bandwidth, intermittent connectivity, and unreliable connectedness of mobile clients. Combat units require connectivity of smart devices that is resilient to the dynamic changes of network topology yet can still provide timely dissemination of communications and intelligence so that warfighters may succeed on the battlefield. This thesis aims to create a communications network of smart devices, using their embedded Bluetooth communications capability. This thesis tests the throughput of the system at the maximum connection distances between devices. It explores the systems capability to properly process and forward network and communications traffic. Lastly, it evaluates the system's ability to adjust to device connection loss while maintaining connections already established and rebuilding connections with devices within connectivity range. The developed application offers a communications network that adapts to device loss by adjusting the network topology while still providing the users with real-time chat capability of all locally available devices.

14. SUBJECT TERMS Infrastructure-less, mobile, network, Bluetooth, scatternet, real-time chat, Android, smart device			15. NUMBER OF PAGES 77
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UU

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2–89)
Prescribed by ANSI Std. 239–18

THIS PAGE INTENTIONALLY LEFT BLANK

C2 AT THE EDGE: OPERATING IN A DISCONNECTED LOW-BANDWIDTH ENVIRONMENT

Alexander J. Beachy
Captain, United States Marine Corps
B.A., Purdue University, 2004

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 2015

Author:	Alexander J. Beachy
Approved by:	Gurminder Singh Thesis Co-Advisor
	John Gibson Thesis Co-Advisor
	Peter Denning Chair, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

A practical solution to providing effective command, control, and communications (C3) information from the warfighting commander to the edge of the battle space involves using light-weight, handheld devices connected on wireless networks; however, infrastructure-based wireless networks, especially at the edge, are characterized by wildly fluctuating bandwidth, intermittent connectivity, and unreliable connectedness of mobile clients. Combat units require connectivity of smart devices that is resilient to the dynamic changes of network topology yet can still provide timely dissemination of communications and intelligence so that warfighters may succeed on the battlefield. This thesis aims to create a communications network of smart devices, using their embedded Bluetooth communications capability. This thesis tests the throughput of the system at the maximum connection distances between devices. It explores the systems capability to properly process and forward network and communications traffic. Lastly, it evaluates the system's ability to adjust to device connection loss while maintaining connections already established and rebuilding connections with devices within connectivity range. The developed application offers a communications network that adapts to device loss by adjusting the network topology while still providing the users with real-time chat capability of all locally available devices.

THIS PAGE INTENTIONALLY LEFT BLANK

Table of Contents

List of Figures

List of Tables

THIS PAGE INTENTIONALLY LEFT BLANK

List of Acronyms and Abbreviations

API	application program interface
C2	command and control
C3	command, control, and communications
CM	Connection Manager
DLCI	Data Link Control Identifier
DOD	Department of Defense
ERTM	Enhanced Retransmission Mode
GUI	graphical user interface
HCI	host controller interface
IEEE	Institute of Electrical and Electronics Engineers
IP	Internet Protocol
ISM	industrial, scientific and medical
LAN	local area network
L2CAP	Logical Link Control and Adaptation Protocol
LMP	Link Manager Protocol
MAC	media access control
MCPeerID	multipeer connectivity peer identification
MEF	Marine Expeditionary Force
OS	operating system
PAN	personal area network

PDU	Protocol Data Unit
QoS	quality of service
RF	radio frequency
RFCOMM	radio frequency communication
SDP	Service Discovery Protocol
SDU	Service Data Unit
SIG	Special Interests Group
TCP	Transmission Control Protocol
TCS BIN	telephony control specification binary
TDD	time division duplex
UDP	User Datagram Protocol
UI	user interface
USB	Universal Serial Bus
UUID	Universally Unique Identifier
WAP	Wireless Application Protocol
WAE	wireless application environment
WPAN	wireless personal area network
XML	Extensible Markup Language

Acknowledgments

I would like to thank my thesis advisors for their guidance and willingness to take on this endeavor. They asked me to kick open the door on this work, and I believe we have done just that. I look forward to continuing this work with them and their future students.

I would like to thank my colleagues Major Scott Fortner, United States Marine Corps, and Lieutenant Seneca Johns, United States Navy, who helped me shape the ideas of the ChatterBox application. Their friendship also helped make the school work enjoyable.

Most importantly, I would like to thank my wife, Victoria, for her constant love and support. She flexed her personal and professional life to allow me to pursue this opportunity. She pushed me when I hit road blocks in my research, writing, and general school work. She gave us the gift of parenthood, and remained the corner stone of our marriage and this family. I am forever grateful to her.

Lastly, to my son, Corbin, always seek to better yourself ... even though I think you are already amazing.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 1: Introduction

As battlefield weaponry and tactics advance, so must the ways in which combat units conduct intra- and inter-unit communications. A practical solution to providing effective command, control, and communications (C3) information from the battlefield commander to the edge of the battle space involves using light-weight, handheld devices connected on wireless networks. Typical wireless networks connected through a cellular infrastructure are difficult to establish at the contested edges of the battle space, costly to maintain, and lack the agility required in very fluid environments. The fluidity at these battle space boundaries is further characterized by wildly fluctuating bandwidth, intermittent connectivity, and unreliable connectedness of mobile devices.

The military has progressed over the last several years toward using tactical smartphones as a means for C3; however, these systems emerged with limitations, including a lack of supporting applications and a reliance on pre-existing military radios as a transmission medium [1]–[3]. In order to build an agile, battlefield-ready, wireless network of smart devices, the network should rely on the embedded wireless capabilities of each smart device (e.g., Bluetooth, Wi-Fi, etc.) and not on the well-established, mobility-constrained infrastructure on which these devices normally communicate. Through the use of the tactical smartphone's embedded communications capabilities, the network can adapt to the users' needs, the types of data being transmitted, and the distances over which communications must occur. This would provide combat forces with a seamless communications system that flexes to the environment as much as they do.

1.1 Purpose

The purpose of this thesis is to build an infrastructure-less communications network of mobile devices through the development of user application software, created for the Android operating system (OS). The thesis includes testing the adaptability of this network to changes in connectivity and network topology typifying the edges of the battlefield. The developed system provides a means of real-time communication by which war-fighting units can communicate across the battle space. This work creates the foundation for a more advanced wireless communications network that incorporates the myriad other embedded wireless technologies.

1.2 Scope and Boundaries

This thesis explores a smart device technology solution as a means of C3 at the edge of the battle space, addressing the challenges posed by implementing such a network in a fluid environment. Although smart devices can use several different wireless communications protocols, this thesis focuses strictly on the embedded Bluetooth capability. This thesis provides a real-time chat application as the starting point for communication across the network, excluding from the research real-time audio or video. The developed application processes and forwards messages and network information from one to all other devices. The established network is built to gracefully handle connection loss, recreating the network topology, as needed, to incorporate all reachable devices.

The battlefield may exist in any clime and place; however, this thesis does not examine the overall effects of weather and terrain on the communications network. All tests conducted occur in a level, controlled, unobstructed communications plane. This thesis does not include the effects maintaining a flexible network has on battery consumption of the individual devices.

Due to limitations in the embedded wireless technologies, network creation, management, and data forwarding must occur at the application layer. This thesis examines the end-to-end throughput and the ordered delivery of data to ensure that the application provides a viable solution. Lastly, the thesis examines the capability of the network to handle connection loss between devices. This thesis shows that the embedded wireless communications capabilities of smart devices can be used to implement a battlefield-ready network of smart devices.

1.3 Thesis Organization

This thesis comprises four additional chapters: background, design and implementation, testing and evaluation, and conclusions and future work. This section outlines each chapter in brief detail.

1.3.1 Background
This chapter provides information on the Bluetooth architecture and previously developed work that bridges a gap between infrastructure-based and infrastructure-less mobile networks. Detailing the inner-workings of the Bluetooth architecture provides clarification as to how the developed software builds, manages, and disseminates information across the infrastructure-less network. It also forms the basis of common terminology used throughout the thesis. Providing information on previously conducted work highlights the advances made—and the limitations that still exist—in this field of study.

1.3.2 Design and Implementation
This third chapter details the individual elements of the software that was created. It provides information on the multi-threaded framework used in order to build connections between individual devices. Furthermore, it shows how the software manipulates these individual connections and implements various timing schemes in order to build a network that can both forward data and react to connection loss.

1.3.3 Testing and Evaluation
This chapter describes the three tests that evaluate the individual connections between devices and the larger network. The first test stresses the individual connection between two devices, at the maximum sustainable connection distance, in order to understand the application end-to-end throughput. Since all forwarding must occur at the application level, the second test examines the ability of the application to properly forward—in order—transmitted information across various network topologies and sizes. The last test manipulates different network topologies in order to evaluate the system's ability to recover from connection loss between devices.

1.3.4 Conclusions and Future Work
The last chapter summarizes the work and findings of this thesis. It also provides suggestions for future research and development work. The suggestions made will help build a more robust and capable infrastructure-less network that has the potential to benefit the military and civilian world for years to come.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 2: Background

This thesis explores the use of Bluetooth to create a power-aware, multi-hop personal area network (PAN) that exchanges messages between members of a group. An understanding of these main features and the work of other application developers will aid in the provide the framework for the decisions and implementations made herein.

2.1 Bluetooth

Swedish mobile-phone manufacturer Ericsson developed Bluetooth in 1994 as a means of connecting devices through a short-range radio link [4]. The Institute of Electrical and Electronics Engineers (IEEE) originally maintained the standard and continues to maintain the personal area network standards of 802.15, but today, the Bluetooth Special Interests Group (SIG), an organization with over 25,800 industry members, monitors and maintains Bluetooth standards.

Bluetooth operates in the unlicensed industrial, scientific and medical (ISM) 2.4 GHz band using a spread spectrum, frequency hopping, full-duplex signal [5]. Bluetooth can connect up to eight devices, forming a piconet, within a range of 10 meters to more than 100 meters, depending on the class of device. Two or more piconets can connect together, forming a scatternet. This can allow the creation of localized, ad-hoc networks of up to 80 devices [4]. The Bluetooth protocol stack stands as the crucial element that provides this networking and data transmission capability of Bluetooth-enabled devices.

2.1.1 Personal Area Networks

IEEE 802.15 [6] covers the details of wireless PAN, under which Bluetooth falls. It defines PANs as a network controlled by a single person or a family [4]. The original standard, 802.15.1, initially defined the Bluetooth specifications; however, the Bluetooth SIG now sets the standards for Bluetooth, and 802.15 covers other methods of developing PANs beyond Bluetooth. In the world of Bluetooth, PANs comprise piconets and scatternets.

Piconets

The piconet rests as the basic unit for modeling a Bluetooth-defined personal area network. The piconet consists of one master device connected to at least one, but up to seven slave devices (see Figure 2.1 for various sized piconet examples).

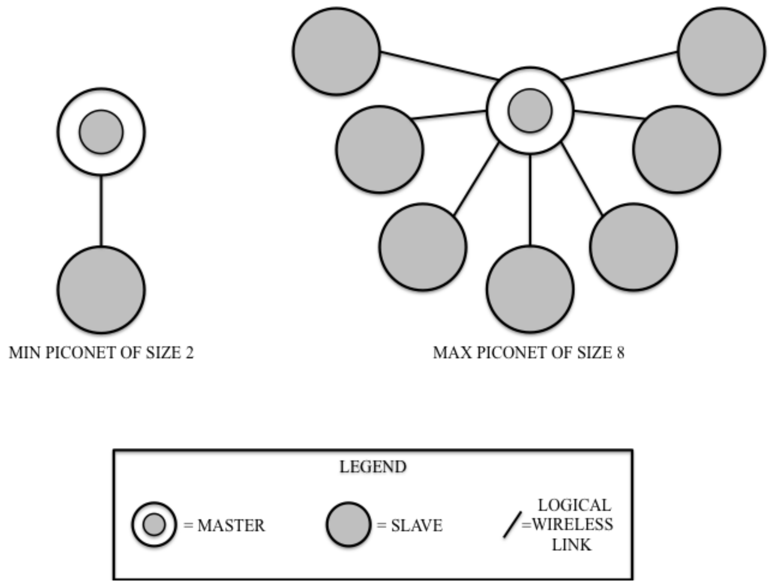


Figure 2.1: Various piconet sizes

The master device will dictate the frequency-hoping sequence and time-phase offset based on its own 48-bit address, as mentioned in Section 2.1.2. All devices of the same piconet operate on the same hop set and offset. Upon initial connection, the master sends its clock count to the slaves who then create an offset which they add to their native clocks for synchronization [7]. With a hop rate of 1600 hops per second, the physical channels are divided into sequentially numbered, time slots of 0.625 μs [4]. The 27 most significant bits of the master clock determine the time slot numbers, ranging from 0 to $2^{27} - 1$.

Bluetooth uses time division duplex (TDD) to determine when the master and the slave transmit data. The master will transmit data on even numbered time slots, and slaves transmit on odd numbered ones. Slaves can transmit only data to the master, but a master can transmit data to any or all slaves. Figure 2.2 depicts a scenario where multiple slaves exist on a single piconet. In the event of multiple slaves on a single piconet, a slave will transmit data only if the master had addressed it directly during the preceding even time slot. All slaves will listen for the master to address them, during even time slots. If the master does not address a slave, then that slave will sleep until the next even time slot before listening again. The master may not transmit any data during an even time slot (see Figure 2.2, time slot $f(k+8)$), in which case no slaves will transmit data during the proceeding odd time slot. The master may do this when it has no data to send or it has commitments in another piconet.

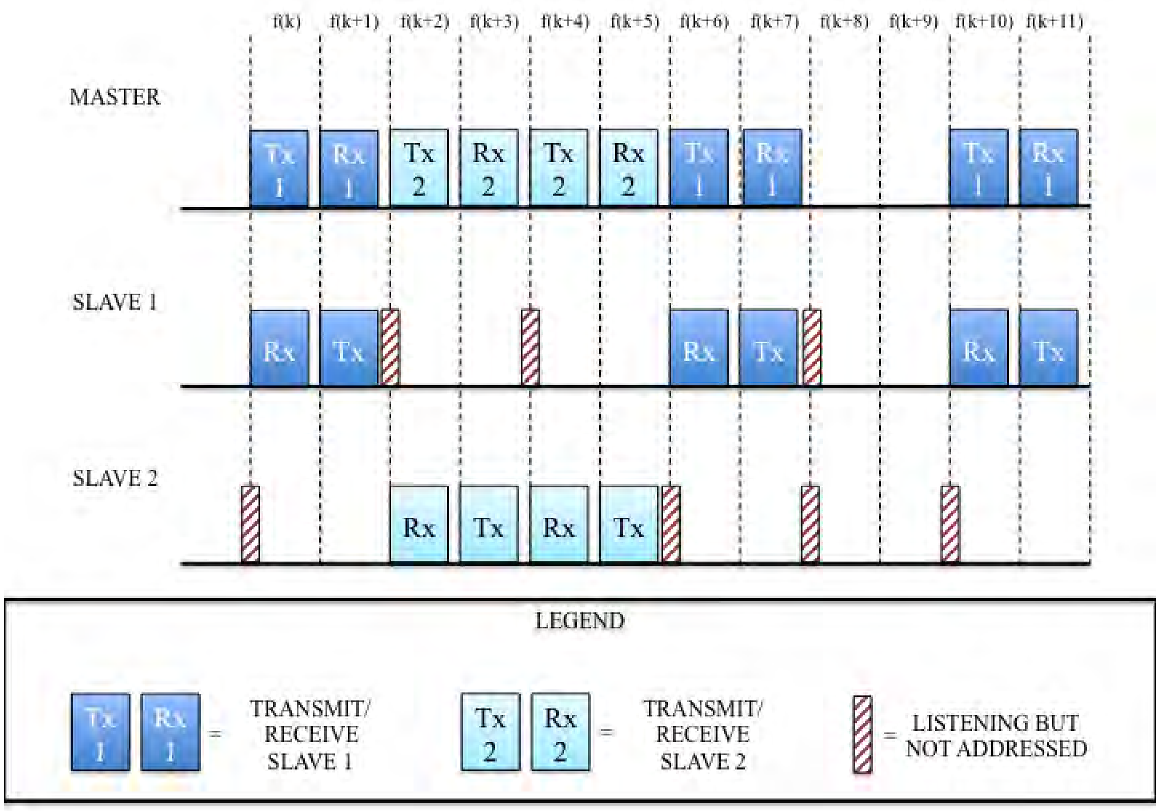


Figure 2.2: Master and multi-slave communications timing, from [7]

Figure 2.3 depicts the Bluetooth TDD system, where $f(k+n)$ indicates the frequency used at time slot $k+n$. Note that once a master or slave transmits, it may continue to do so for one, three, or five time slots. If a device requires more than one time slot to complete a transmission, it will use the same frequency throughout the transmission, bypassing other intended frequencies of the hop sequence. When the transmission completes, the hop sequence will resume on schedule, skipping the either two or four hop frequencies that would have occurred during that transmission time.

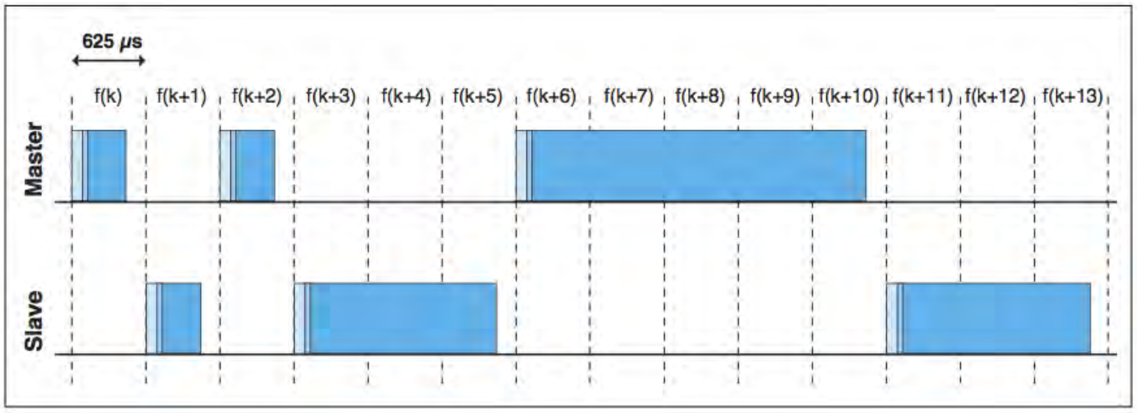


Figure 2.3: Bluetooth TDD time slots, from [7]

Scatternets

Multiple piconets can reside in the same geographic location. When a device participates in two or more piconets, the combination of those piconets creates a scatternet. A single device can serve as a slave for multiple piconets; however, it can only ever serve as a master for one piconet. Since each piconet has its own master, each will have its own hop set and phase. A device that operates within multiple piconets will use time multiplexing in order to communicate on the appropriate channel and phase for each piconet. Figure 2.4 demonstrates the scenario where a single device serves as both a master of one piconet and a slave of another, where $f(k+n)$ and $g(k+n)$ are the respective hop set time slots for each piconet. Note that the phases of the two piconets do not align; therefore, when the device transitions from one piconet to the other it must wait to transmit or receive data until the appropriate time slot occurs. This mean that although the time slot $g(k+9)$ is free for transmission the device cannot transmit during that slot, rendering the time slot unuseful, because it did not receive a packet addressed to it from the master in the preceding time slot; therefore, it must wait until addressed in time slot $g(k+10)$ before sending data in the proceeding time slot, $g(k+11)$. Although this methodology creates inefficiencies in the use of the physical layer, it provides the means necessary for a series of piconets to become a scatternet.

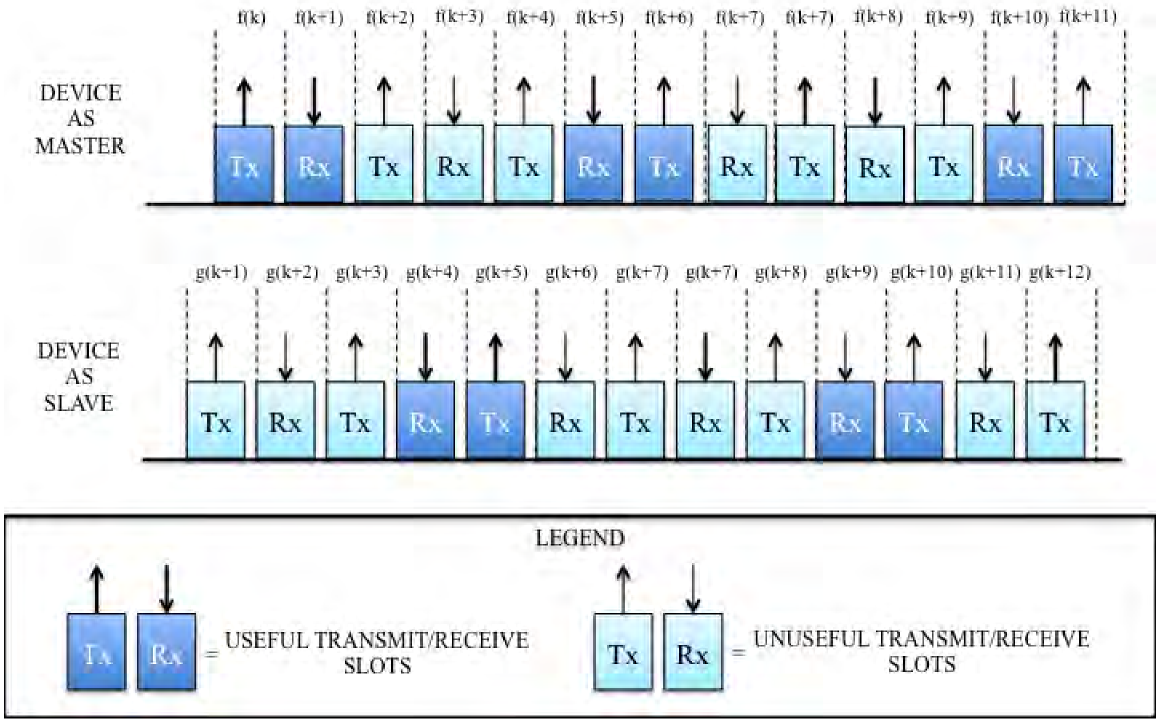
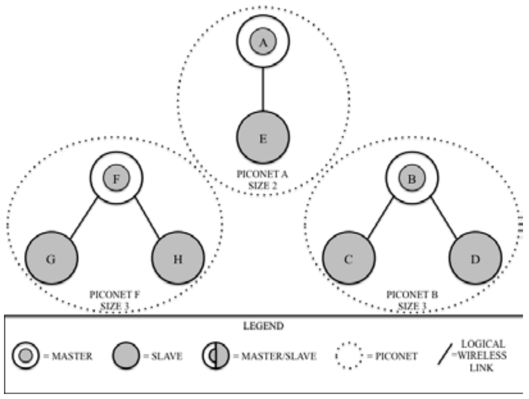


Figure 2.4: Master-slave device useful and unuseful time slots, from [8]

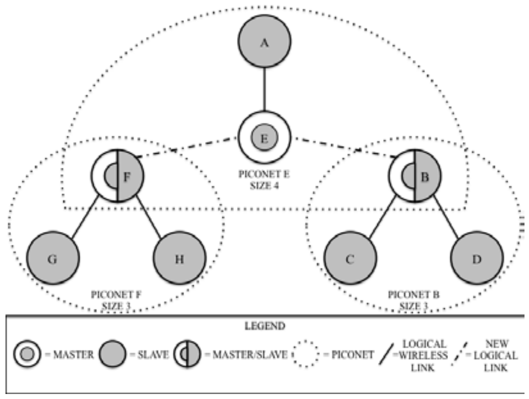
The inter-device, master-slave roles will switch during the creation of scatternets. In the case where a slave of one piconet creates a connection, as the master, with another piconet, a role switch will occur with that slave's original master. Figure 2.5 depicts just such an occasion: Figure 2.5(a) shows three piconets, where devices A, B, and F are the masters of E, C and D, and G and H, respectively. When slave device E forms a connection with master devices B and F (as depicted by the new logical link lines), devices B and F assume the roles of both master and slave; furthermore, a role switch occurs between A and E. Figure 2.5(b) shows the role transitions upon the completion of building the scatternet.

Figure 2.6 shows a similar role reversal scenario, but in this instance device A has several slaves, not just device E. During the building of the scatternet, by way of device E creating connections to devices B and F, devices A and E will still perform a role switch, but device

A will remain the master of the other nodes in its original piconet, devices Y and Z [7].

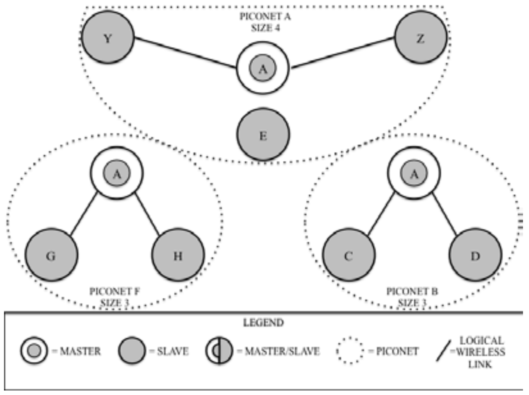


(a) Three piconets before scatternet

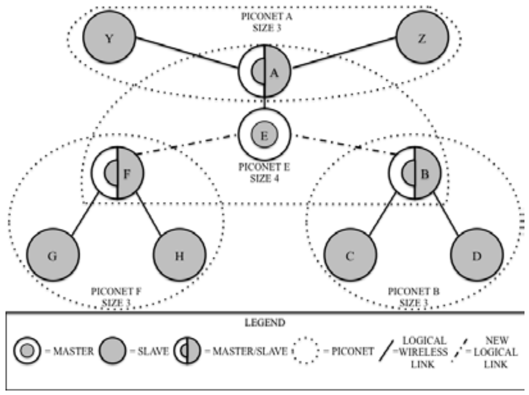


(b) Scatternet created from three piconets

Figure 2.5: Piconets before and after the creation of one scatternet



(a) Piconets before scatternet and role switch



(b) Scatternet after A-E role switch

Figure 2.6: Role reversal after the creation of a scatternet

2.1.2 Protocol Stack

The Bluetooth specification [7] defines the protocol stack in terms of core and profile specifications. The core specifications detail the use of several data-link control protocols and the physical layer within the protocol stack. The profile specification outlines various features of each layer—both core and non-core—in the protocol stacks based on different usage models of the Bluetooth system. The profiles define the interactions between layers on a specified device and the peer-to-peer interaction of each layer across devices. These standardized profiles provide interoperability between different system manufacturers whose products perform the same function (e.g., headset linkage, internet bridge, file transfer, etc.). Figure 2.7 depicts a generic Bluetooth profile separated into the host and primary controller portions of the stack. The various boxes represent the different types of protocols (core, cable replacement, service, and adopted) and the interaction between the layered protocols.

The Bluetooth specifications divides the stack into upper and lower layers, delineated by a host controller interface (HCI). The higher layer protocols, also known as the Bluetooth Host, manage communications between the software application and the primary controller. The Bluetooth Host comprises a single core protocol, Logical Link Control and Adaptation Protocol (L2CAP), a service protocol, Service Discovery Protocol (SDP), and many other potential protocols: the cable-replacement protocol, radio frequency communication (RFCOMM), the telephony-control protocol, telephony control specification binary (TCS BIN), and other adopted protocols that the Bluetooth specification does not detail (e.g., Wireless Application Protocol (WAP), Transmission Control Protocol (TCP), Internet Protocol (IP)).

The lower-layer protocols, also known as the Primary Controller, manage the inter-device connections [5]. It comprises several protocols, including three core protocols: Link Manager Protocol (LMP), baseband, and the physical radio. These protocols offer three basic services: two control-plane services (device and transport control), and a single user-plane service (data service). The device control service provides a means for modification of the Bluetooth device's behavior and modes, while the transport control service facilitates the creation, modification, and release of channels and links. The data service provides for the submission of data and its subsequent transmission across the designated channels.

Although there exist many protocols and protocol profiles, the development of handheld device communications require several essential protocols: RFCOMM, L2CAP, and baseband. RFCOMM emulates a serial port for the application and multiplexes it for L2CAP. L2CAP converts the data into specific sized data units which it then transfers to the baseband for transmission across the physical medium.

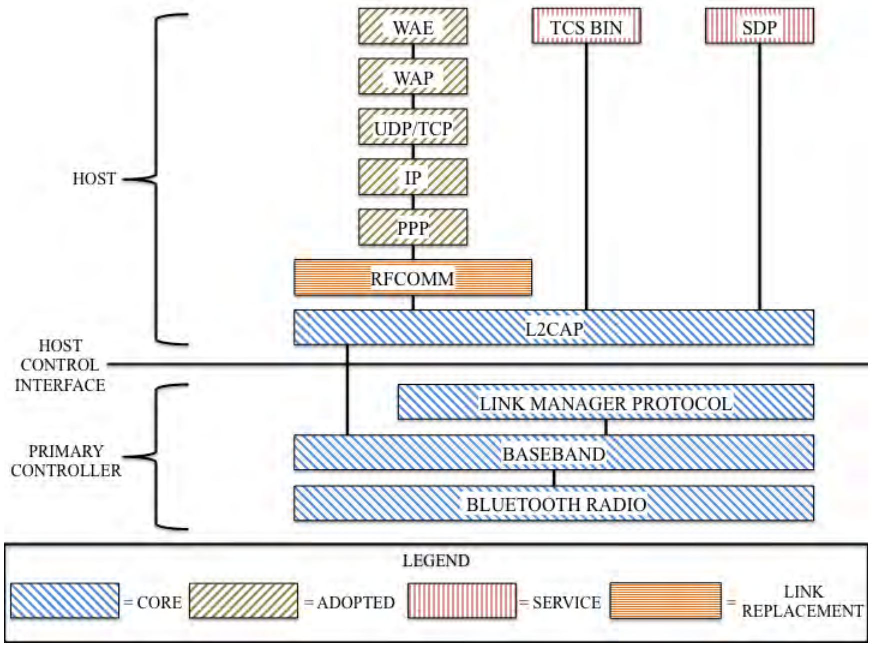


Figure 2.7: Specified Bluetooth Protocol Stack, from [4]

RFCOMM

Android applications use the RFCOMM protocol to provide an emulated serial port over L2CAP through which the application sends and receives data. It supports up to 60 simultaneous connections between two Bluetooth connected devices; however, actual implementation varies based on the application specific requirements. RFCOMM identifies a connection between devices based on its Data Link Control Identifier (DLCI), a six-bit identification code. RFCOMM assigns a value from 2 to 61 for the unique connections; it uses DLCI 0 for the control channel, while DLCI 62 and 63 are reserved and DLCI 1 is unusable [9].

Figure 2.8 depicts the Bluetooth-enabled, Android application reference model for RFCOMM. The application will request a port from the RFCOMM protocol; the Port Emulation Entity will link the application program interface (API) to this RFCOMM service. In order for an application to connect and share data, it must use this RFCOMM architecture.

RFCOMM multiplexes the various emulated serial ports, providing a data stream and control channel to the L2CAP [9]. RFCOMM relies upon L2CAP to establish a connection

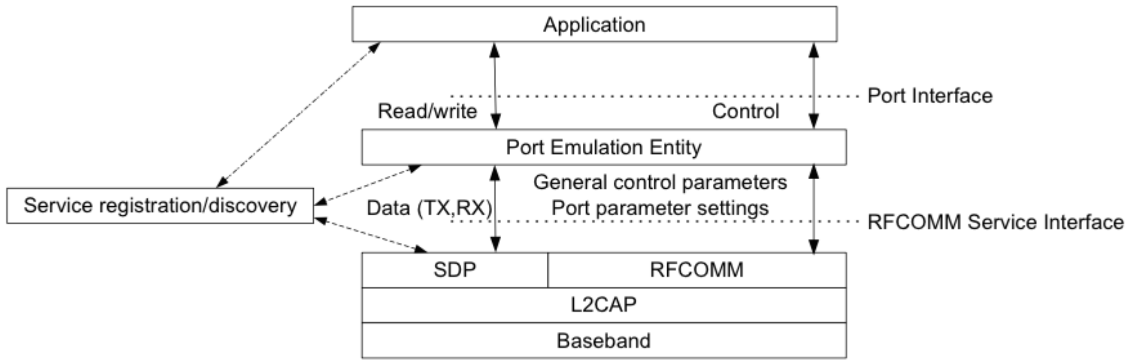


Figure 2.8: RFCOMM Reference Model, from [9]

with another RFCOMM entity on the other side; furthermore, it relies on L2CAP to provide non-duplicated data in the correct order across the connection.

L2CAP

L2CAP is the bridge between the Bluetooth Host and the lower layer protocols. As such, it manages resources when submitting data to the baseband for transmission across the link. L2CAP segments and order the application's Service Data Units (SDUs) into manageable sized Protocol Data Units (PDUs). Based on the size of the controller buffer, it will fragment the PDU into start and continuation packets for optimum use of the buffer space [7]. At the receiving end, L2CAP orders and reassembles the PDU and subsequent SDU for higher layer protocols. L2CAP basic mode allows for three packet size configurations: 48-byte minimum sized payload; 672-byte, default-size payload; and 64-kilobyte, maximum-size payload [5].

In order to ensure that L2CAP channels with quality of service (QoS) commitment have access to the physical channel, L2CAP manages scheduling between controller buffers and the designated channels. The limited controller buffer size and the finite bandwidth of the host drive the design of L2CAP's controller buffer management. The L2CAP resource managers also ensure that the host applications submit SDUs within the negotiated QoS settings [7].

L2CAP also provides the Bluetooth protocol stack with error detection and retransmission of PDUs [7]. recommends this feature for "applications with requirements for a low probability of undetected errors in the user data." L2CAP further provides a window-based flow control, similar to that of TCP, to manage the receiving device's buffer allocation. Recent enhancements to L2CAP provide two new modes of operation: a streaming mode which does not provide re-transmission or flow control, similar to streaming data over User Datagram Protocol (UDP); and an Enhanced Retransmission Mode (ERTM) that offers improved performance over its predecessor.

Baseband

The baseband layer of the protocol stack prepares the data—packet format and addressing, power requirements, and timing—for access to and transport across the physical radio medium. In order to properly provide access to different radio access requests, the baseband first negotiates an access contract for each request. This access contract serves as a QoS agreement so that the user application may operate properly under a certain expected performance level. Once the baseband has negotiated the access contract, it will then grant radio medium time slots for each application based on the respective QoS agreement [7].

The baseband defines a physical channel by using a pseudo-random radio frequency (RF) channel-hopping sequence along with the allocated time slot and an access code. It uses the 48-bit Bluetooth device address as a seed for the pseudo-random RF hopping sequence. The baseband uses the Bluetooth clock to determine the phase of the hopping sequence. Section 2.1.1 details the coordination of this information between the master and slave Bluetooth devices.

The baseband has a subcomponent, the device manager, that controls the general behavior of the Bluetooth device. This device manager requests access to the physical medium through the baseband so that it can search for available devices, connect to other devices, and enable discoverability and connect-ability of the local device. The device manager can also alter the device name and manage other basic device Bluetooth functionality [7].

2.2 Previous Work

Advances in Mobile Mesh Networks have led to the creation of proprietary text-based communications applications. Bluetooth and Wi-Fi Direct enable these types of applications to communicate without the use of infrastucture-based connectivity [10]. Apple has developed their own API which incorporates the use of standard Wi-Fi, Wi-Fi Direct, and Bluetooth technology to allow application developers to send "message-based data, streaming data, and resources (such as files)" between mobile devices running an application that uses this API [11]. FireChat, created by Open Garden, emerged in March 2014 as a proprietary, infrastructure-less, text-messaging application [10].

2.2.1 Apple's Multipeer Connectivity

The Multipeer Connectivity API provides applications with the ability to create communication sessions between the same application running on different devices. The API works in two phases: discovery and session. During the discovery phase, when a user runs an application, it will advertise basic information about the device and the types of sessions it supports. Also during this phase, a user can search, through a graphical user interface (GUI), those users advertising a specific type of session capability. The searching user can then request other users to join a particular session. Upon a user accepting an invitation, a session is created. While established as a member of a session, a device can communicate with one or more members of the session (as this API is proprietary, the specific details of how it forwards traffic is unknown). A session maintains a set of multipeer connectivity peer identifications (MCPeerIDs)—a combination of the application and device's unique identification codes. The application will use this MCPeerID for communication purposes and also to notify users when a member enters or leaves a session [11]. Apple maintains Multipeer Connectivity as a proprietary API, and Android devices do not currently have their own version of this technology, making interoperability between device types more difficult.

2.2.2 FireChat

FireChat uses Bluetooth, Wi-Fi Direct, and Multipeer Connectivity to connect similar type devices running the application within a local area (within a typical maximum range of 40-70 meters), also known as nearby mode. FireChat, when operating in nearby mode, enables directly connected devices within a chatroom to share text messages [10]. This feature works only with devices in close proximity, and it does not advertise multi-hop routing of messages in nearby mode [12]. The development of FireChat has paved the way for new types of applications; however, it has also introduced its own set of advantages and disadvantages.

A key advantage to ad-hoc networks, such as FireChat in nearby mode, stems from the lack of infrastructure, which means that government agencies and controlling bodies cannot monitor nor collect message traffic. It also means that when an organization or a natural disaster prevents the normal flow of data, users still maintain the ability to communicate [10]. These advantages allowed protestors of the Chinese political system to communicate through FireChat, anonymously, without government intervention or monitoring [13].

The greatest disadvantage to the FireChat implementation is an inherent lack of security: [14] identified that the FireChat app does not allow for private messaging, nor does it encrypt its data prior to transmission. Although large organizations may not have the ability to monitor all messages, and a user can create a seemingly anonymous username, an attacker could listen in on local conversations and gain potentially private information from unsuspecting users.

As discussed in Section 2.2.1, the proprietary nature of connection methods prevent Apple FireChat users from connecting and communicating with Android FireChat users, when operating independent of internet connectivity [15]. This greatly limits the scalability of such a communication method. This complication comes from FireChats reliance on the Multipeer Connectivity framework for Apple users.

2.3 Summary

This chapter discussed the basic building blocks of Bluetooth. Building a piconet requires an understanding of Bluetooth protocol stack, specifically how devices connect and distribute data. In order to build an Android application that allows a scalable number of devices to share information without any infrastructure, it is important to understand how piconets can combine to build a scatternet. Lastly, this chapter examined proprietary software that uses Bluetooth and other standards to allow devices to share information. Chapter 3 will build upon this knowledge in the development of open-source software that uses Bluetooth scatternets to allow multi-hop sharing of data across Android devices so that members of small tactical units may communicate without established infrastructure.

CHAPTER 3:
Design and Implementation

The use of smart devices on the battlefield provides ground combat forces the ability to use several different wireless communications protocols—Bluetooth, WiFi, and other network technologies—to create ad-hoc networks. With the ability to share data across these varying protocols simultaneously, the smart device can bridge data received from one protocol onto another, creating a more robust, infrastructure-less network. Figure 3.1 abstracts the idea of bridging Bluetooth scatternets with WiFi and other wireless protocol networks. This bridging concept gives warfighters the flexibility to move about the battlespace, while maintaining network connectivity, unaware of the shifting between wireless protocols that occur as the distance and throughput requirements change.

This thesis will build the first stage of this overall network design using the Bluetooth scatternet to build and maintain the network. It will do so through a user application running on the Android OS. This application will use the Bluetooth satternet to provide warfighters with an ability to communicate via chat messaging to all participating members. Chat messaging, also known as chat, allows ground combat forces to track accurate communications and log the chat messages to be reviewed at a later time.

3.1 System Description

This thesis uses Java and Extensible Markup Language (XML) programming to create an infrastructure-less Android chat application, named ChatterBox, that uses Bluetooth to share messages. The ChatterBox application gets its name as a play on words of an extremely talkative person; however, it also comes from the concept of a three-dimensional space in which conversations, or chatter, can occur. ChatterBox provides text-based communications for small unit warfighters (e.g., a platoon of infantrymen) through the use of smart devices and their embedded radio capabilities. The application runs on smart devices that use the Android operating system, and it uses the device's Bluetooth radio to build a communications network for the devices. Smart phones and tablets are the intended device for this application as infantrymen can easily carry these devices, and the Department of Defense (DOD) actively pursues a tactical smart device for military operations. The design

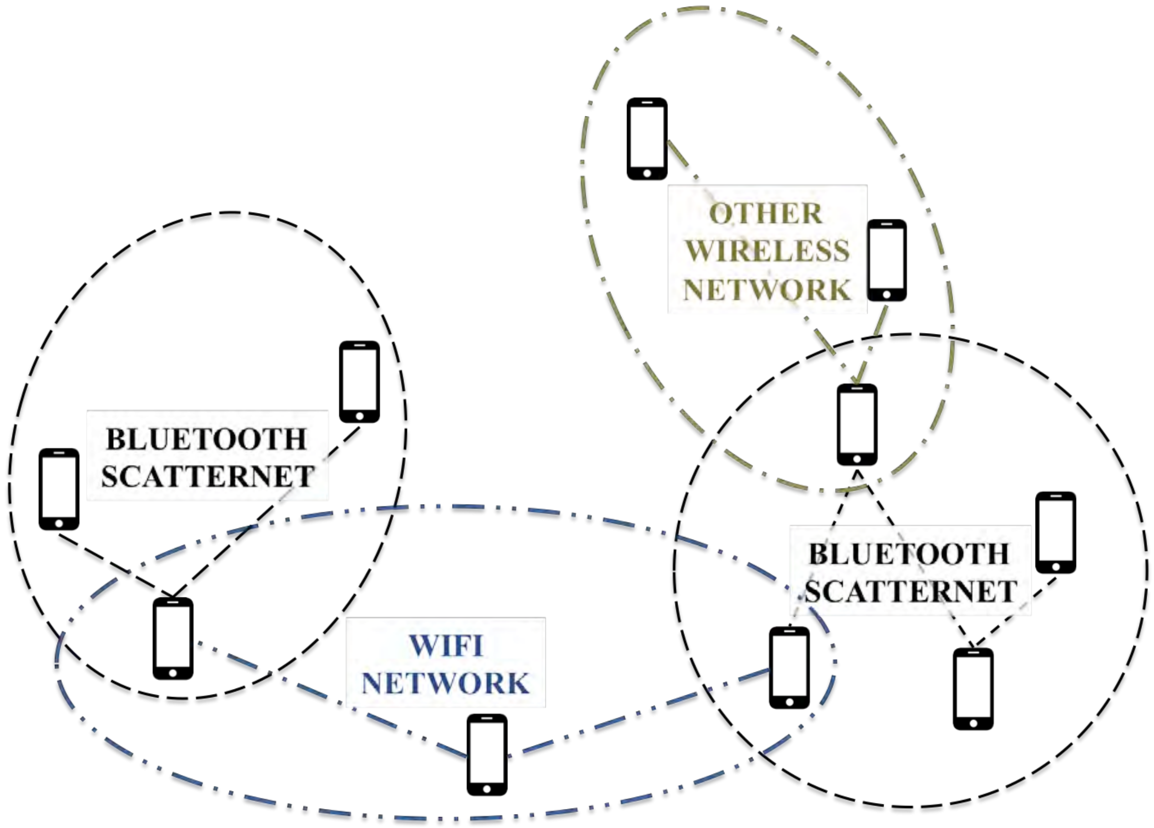


Figure 3.1: Concept behind infrastructure-less communications using inherent Smart Device capabilities

of this application must allow a warfighter, one with little or no experience, to create a connection between devices and begin communicating, with ease.

3.2 Application Description

ChatterBox provides the user with three screens in order to carry out the task of building and communicating across a network: a start screen, a device selection screen, and a chat screen (Figures 3.2, 3.3, and 3.4, depict the user interface (UI) screens for each, respectively). The start screen prepares the device for initial Bluetooth operations. The device selection screen provides the user with a list of Bluetooth devices from which the user can select. The chat screen works as the UI for the last and most important activity and modules of the application. The backbone to this last screen provides the framework for building and managing connections, as well as, sending and processing the various messages.

3.3 Start Screen

The start screen performs three basic, but essential, tasks before a Bluetooth connection can be made: verify that the device has Bluetooth capability, establish a username, and verify that Bluetooth is turned on. Figure 3.2 shows the feedback that the user will get at the beginning of every step in this setup process.

Bluetooth Support The start screen must first verify that the device supports Bluetooth, (i.e., it has Bluetooth capability). Although most current devices support Bluetooth, if it is not installed or is malfunctioning, then this simple check prevents the application from attempting to access something that does not exist, thereby crashing the whole system. In the event that the device does not have Bluetooth capability, the app will shut down gracefully while preventing harm to the overall device.

Username The second step in the start screen's setup process is to prompt the user for a username. When the user submits a name—a minimum of one character is required—the application sets the provided name as the device's Bluetooth name. This makes the Bluetooth device easier to identify by humans during the search for local Bluetooth devices (see Section 3.4 for details). Although smart device's default the Bluetooth device name to the device make or model (e.g., SAMSUNG-SGH-I747 for the Samsung Galaxy SIII), this step did not exist during the initial ChatterBox versions because the initial implementation only used four devices, making it relatively easy to discern one device from another. As the implementation progressed to eight devices, and with the realization that this application could scale quickly, the importance of a username in identifying one device from another became paramount. Since the devices do not share a common link and since there is no infrastructure to verify a unique username, two users could have the same username; however, this application is designed for the small-unit warfighters with the expectation that their standard operating procedures will ensure that two members of the same unit do not share the same username.

The application also uses the name when creating ChatterBox messages (see Section 3.5.1 for more detail), which the application will use to display in the chat screen (3.5) and use when storing messages to the device's internal storage. Furthermore, the application stores the name as a saved preference. The application will fill in the username with this saved

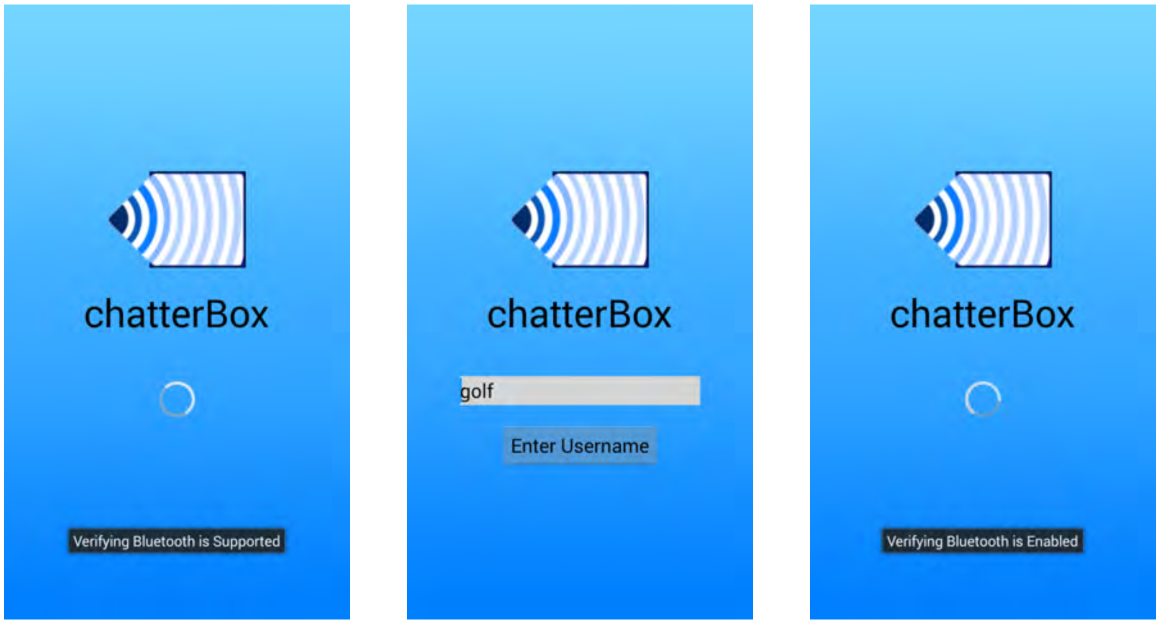

chatterBox
Verifying Bluetooth is Supported
chatterBox
golf
Enter Username
chatterBox
Verifying Bluetooth is Enabled

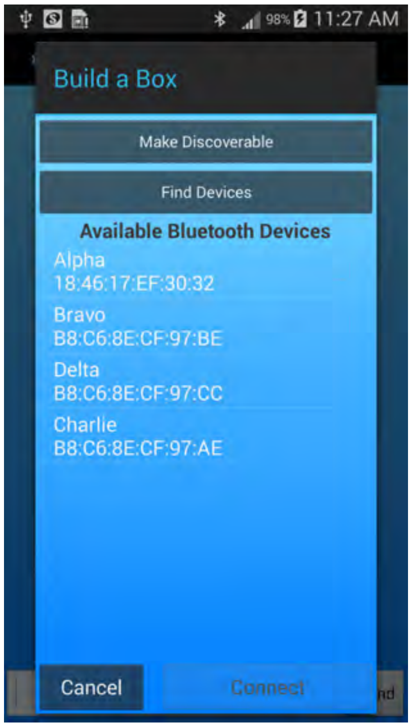
Figure 3.2: Start screen setup progression

preference so that the user does not have to retype the username every time he or she starts ChatterBox. If the user changes the username from what is filled in by the saved preference, the new username will replace the old one as the saved preference. The username is not intended as an authentication measure, merely as a way to identify one device from another.

Bluetooth Enabled The last step of the setup process ensures that the device's Bluetooth capability is turned on. This will allow the application to operate properly when creating, maintaining, and sharing data across a Bluetooth connection. The Android OS has a limitation within the Bluetooth API: it requires user agreement that ChatterBox can activate the Bluetooth system. Furthermore, the user can deactivate the Bluetooth at any time, rendering ChatterBox unusable during the period where Bluetooth is not running. The application cannot lock the Bluetooth system on, but when Bluetooth becomes active, it can reestablish a connection with the group and continue communicating.

3.4 Device Selection Screen

At the completion of the start screen setup process, the application will display a device selection screen. It begins by searching for discoverable devices within close proximity (approximately less than 40 yards). It then filters the discoverable devices to only smart phones and tablets. Once filtered, it displays a list of the remaining devices' usernames and media access control (MAC) addresses on the screen for the user to select. Figure 3.3 shows the screen shot of a device after finding locally available smart devices.



Available Bluetooth Devices

Alpha
18:46:17:EF:30:32

Bravo
B8:C6:8E:CF:97:BE

Delta
B8:C6:8E:CF:97:CC

Charlie
B8:C6:8E:CF:97:AE

Figure 3.3: Device selection screen with displayed list of smart devices.

Limitation The Android OS prevents an application from making a device discoverable without user agreement, and it only allows discoverability for a set period of time, in the case of ChatterBox this time limit is 300 seconds. Although this works to limit malicious software from broadcasting a device's Bluetooth signature, and it prevents unnecessary power drainage by making a device constantly run in a discoverable mode, it limits the automation of ChatterBox by requiring user interaction every time the application requires enabling the discoverable mode. This means that if a user does not select that his or her device become discoverable, then the searching device cannot find it; however, if the searching device does have a valid Bluetooth MAC address for the device, a connection can still be made. ChatterBox relies on this as a means to create a connection between two devices that have previously never connected, yet at least one is aware of the other in the scatternet.

This is what enables ChatterBox to have resiliency within the scatternet when connections break (see Section 3.5.3 for more information).

Once the user has selected the devices with which he or she would like to connect, the device selection screen creates a list of of each device's MAC address and passes this to the last, and most important, screen, the chat screen.

3.5 Chat Screen

The chat screen comprises the bulk of the operational components of ChatterBox. The screen displays the messages as the device receives them; specifically, it displays the username and message payload from each ChatterBox message (See Section 3.5.1, CBMessages, for details). It also has a drop-down menu that allows the user to select different activities and tests to perform. Most importantly, it hosts the Connection Manager (CM) (3.5.3) module, which handles the multiple threads for establishing and maintaining inter-device connections, creating and parcing the CBMessages, and running throughput and ordering tests.

Figure 3.4 shows an example of what the user will see when sending and receiving messages. Received messages will display the source name and the message; for example, "Delta: checking in." Sent messages will display after a successful transmission, but the username is replaced by "Me," for example, "Me: target acquired." The "SYSTEM" name identifies messages produced by the ChatterBox system. These messages typically announce when a successful connection occurs or fails, when an established connection is lost, and when a connection reattempt fails. These messages stay local to that particular device (i.e., the device does not transmit these messages across the scatternet).

3.5.1 CBMessages

In order for the system to provide a multiple-hop message capability throughout a scatternet (2.1.1), a standardized message format had to be developed. This standardized message, named CBMessage, would provide a device several hops away from the initial source with enough information to process and utilize the transmitted data. The CBMessage design changed frequently throughout the development process; initially, it only had a payload, until multiple-hop capability became a reality. From there, enhancements added a source

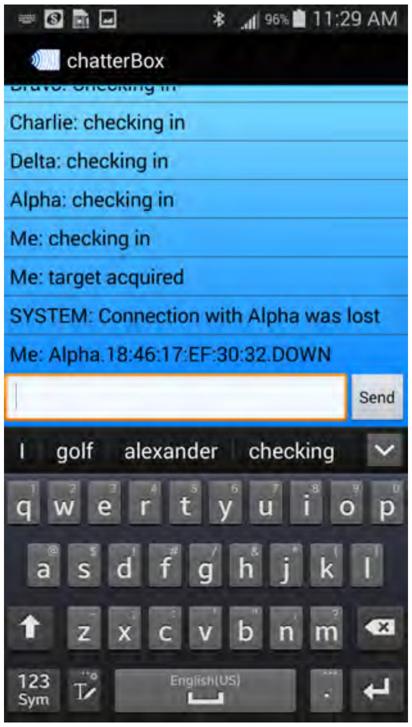

Charlie: checking in
Delta: checking in
Alpha: checking in
Me: checking in
Me: target acquired
SYSTEM: Connection with Alpha was lost
Me: Alpha.18:46:17:EF:30:32.DOWN

Figure 3.4: Chat screen display

name, source and destination MAC addresses, a message time stamp, and a message type.

The CBMessage has a variable length header due to the inclusion of the source name. The source-name header segment must be at least one byte long, but has no upper-size limit; this design allows a war-fighting unit using this application to set a standardized naming convention for each device, without limitation. The remaining header segments are fixed length; however, all header segments are delineated by a single "%" sign. Although this delineation method and the source-name segment increases the header length and overhead for each message, it aids in human readability of the entire message when the message is saved to disc for later access.

The source MAC address comes from the device's Bluetooth MAC address, and is not otherwise changeable through the application. Currently the destination address broadcast MAC address of "FF:FF:FF:FF:FF:FF" is used for all outbound messages as the application design intends for all users within a war-fighting unit to share information quickly without having to reference a given user in particular.

A time stamp is the next element of the CBMessage header. Every message is stamped with a local time for that device, with hundredths of a millisecond precision, just prior to sending. Since the device's do not sync their clocks between one another, each device must maintain a record of the most recent time stamp received from every other device. Other devices use this time stamp to determine the freshness of the message. Section 3.5.3 gives more detail as to how ChatterBox uses the time stamp.

The message type is the last element of the CBMessage header , and it is represented by an integer value. The application currently uses four different message types: text, connection status, throughput test, and ordering test. The text type messages encompass those messages created by one user for the intended purpose of transmitting to the larger group; furthermore, this is the only type where the user can control the message payload, all other types receive the message payload from the local device. The connection status type messages share any change in connection status between two devices. The last two types, throughput and ordering test types, are used for testing and evaluation purposes. Section 3.5.3 provides more information on how the application processes each type of message.

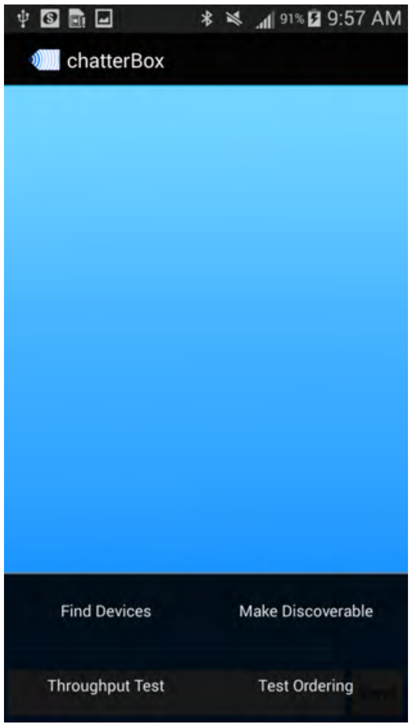
During throughput testing, where the application attempts to transmit hundreds of large messages successively, the receiving device would receive messages faster than it could process, and it would not always delineate one message from the next. When individual messages are transmitted at normal chat-messaging pace, this issue does not arise. In order to ensure that the system could handle any surge in message traffic, a beginning-of-message and an end-of-message delimiters were added to the CBMessage. This allows the application to positively identify when a message starts and ends. Each delimiter is one byte in length. Figure 3.5 depicts the final layout of the CBMessage with respect to each component and how many bytes each component occupies.

Start-of-Message Delimiter (1 byte)	**Device Name (Variable Length)**	**Source Address (18 bytes)**	**Destination Address (18 bytes)**	**Source Time Stamp (8 bytes)**	**Type (4 bytes)**	**Payload (Variable Length)**	**End-of-Message Delimiter (1 byte)**

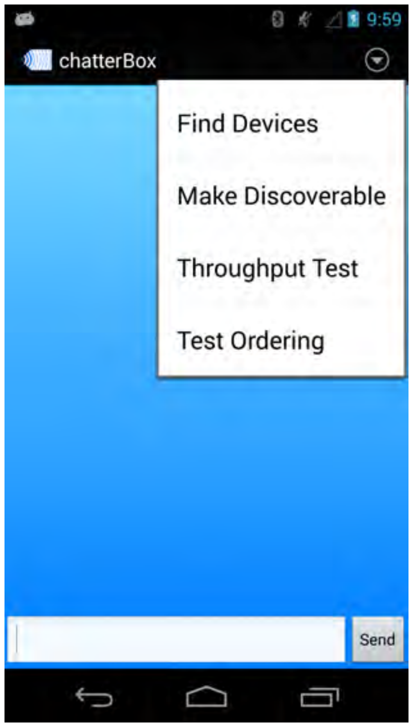
Figure 3.5: Breakdown of CBMessage format

3.5.2 Menu Items

The chat screen has a drop down—or pop-up depending on the model of device (see Figure 3.6 for display variations)—menu that provides the user with the ability to find and connect to local devices, activate Bluetooth discoverability, and launch a series of tests.



(a) Pop-up menu (Galaxy S3)



(b) Drop-down menu (Galaxy Nexus)

Figure 3.6: Menu variations between device models

The Find Devices menu item invokes the device selection screen (Section 3.4). This allows users to continuously add devices to the scatternet as missions change, or as a backup solution in the event that an already established connection breaks but the automatic reconnect does not work. The Make Discoverable menu item works in concert with the previous menu item, as it activates Bluetooth discoverability. The limitations paragraph of Section 3.4 discusses why a limitation in the Android OS requires such a menu item. The last two menu items allow the user to launch throughput and message ordering tests. Chapter 4 will discuss these tests in detail.

3.5.3 Connection Manager

The CM module comprises the heart of the networking and message transfer and processing capability of the ChatterBox application. It creates and manages multiple threads that listen for connection requests, create connection requests, and maintain inter-device connections. It processes each received message, verifying it is a new message, saving it, displaying it, and forwarding it, as required. It also runs the throughput and message ordering tests discussed in Chapter 4.

Connection Management Threads

The connection management section of the CM comprises three different threads: a listen thread, which listens for Bluetooth connection requests from other devices; a request thread, which requests a connection with another device; and a connected thread, which sends and receives CBMessages. The CM always maintains exactly one listening thread; however, it may create and manage multiple request and connected threads depending on the size of the piconet or scatternet and the state of connections, remembering that each device can connect to, at most, seven other devices.

Listen Thread Upon the successful completion of the start screen setup phase, the CM for each device will create a thread which listens for an incoming Bluetooth connection request, a Listen Thread. Upon instantiation of the thread, it creates a Bluetooth RFCOMM socket by which it can listen for requests. The thread provides the Bluetooth API with a 128-bit Universally Unique Identifier (UUID), a value unique to the ChatterBox app across all devices. The API will build the connections down the Bluetooth protocol stack. If the system successfully establishes an RFCOMM socket for this UUID, then the thread will start to listen for requests that pertain to only that UUID. If the system fails to establish and RFCOMM socket, it will return an error. From there the CM will destroy that thread and begin a new one, allowing the device to listen for another connection attempt.

Request Thread When the user selects a series of devices from the device selection screen, the application sends a list of those devices to the CM. The CM will create Bluetooth device objects for each selected device. It will then create a thread for each device, which requests a connection to that given device. Upon instantiation, the thread will request an RFCOMM socket in similar fashion to that of the listening thread, providing the same UUID. The

OS will then begin establishing a connection between the two devices using the layers below RFCOMM in the protocol stack (see Chapter 2 for details). If the system creates a successful connection between the two devices, it will return an established RFCOMM socket to the request thread. If the connection is unsuccessful, the thread will receive an error from the system, which will then prompt the CM to terminate the thread and reattempt the connection (see Section 3.5.3 for details on how the CM handles connection failures). If successful, however, the thread will return this socket to the CM, which the CM will use to create the final thread in the connection process, the Connected Thread.

During implementation, if the CM attempted to connect to four or more devices, first time connection success averaged 75 percent. In order to improve first time connection success, a delay of 100 nanoseconds was added between the connection request for each device (e.g., the time between connection requests for the first device and third device would be 200 nanoseconds). This delay improved the first time connection success to 99 percent, without causing any noticeable application lag to the user.

Connected Thread Once the Listen Thread and the Request Thread have both established a successful RFCOMM socket and provided it to the CM, the CM will then create a Connected Thread, passing the socket as a parameter. The Connected Thread will use the RFCOMM socket to create buffered input and output streams. These streams allow the thread to send and receive data. Before a CBMessage is transmitted, it is converted to a byte array format so that the output stream can properly send the data. On the receiving side, when the input stream receives data, the thread takes all data in the buffer, converts it back to its original CBMessage format and separates the individual messages, in the event that multiple messages were in the buffer. With the messages separated, the thread passes them to the CM for processing.

Thread Interaction Figure 3.7 depicts the interaction between these threads on a single device and between shared devices. This is an overview of the application, and it does not show the interactions that occur on the Bluetooth protocol stack. Device A acts as the master and Device B as the slave. When Device B advertises itself to Device A (1), Device A creates a Bluetooth device object which it then provides to the Request Thread (2). The Request Thread creates an RFCOMM socket and uses the address of the Bluetooth device object to send a connection request, which includes the ChatterBox UUID, through the socket to Device B (3). Meanwhile, Device B will have created an RFCOMM socket via the Listen Thread, providing the same UUID. When Device B receive's Device A's request with the UUID, Device B will verify the UUID (this occurs below the user-application level) and the two devices will establish a connection between sockets (4). From that connection, both devices independently pass their established RFCOMM socket to the Connected Thread, via the CM (5). Both devices will then share data through the RFCOMM socket via the Connected Thread (6).

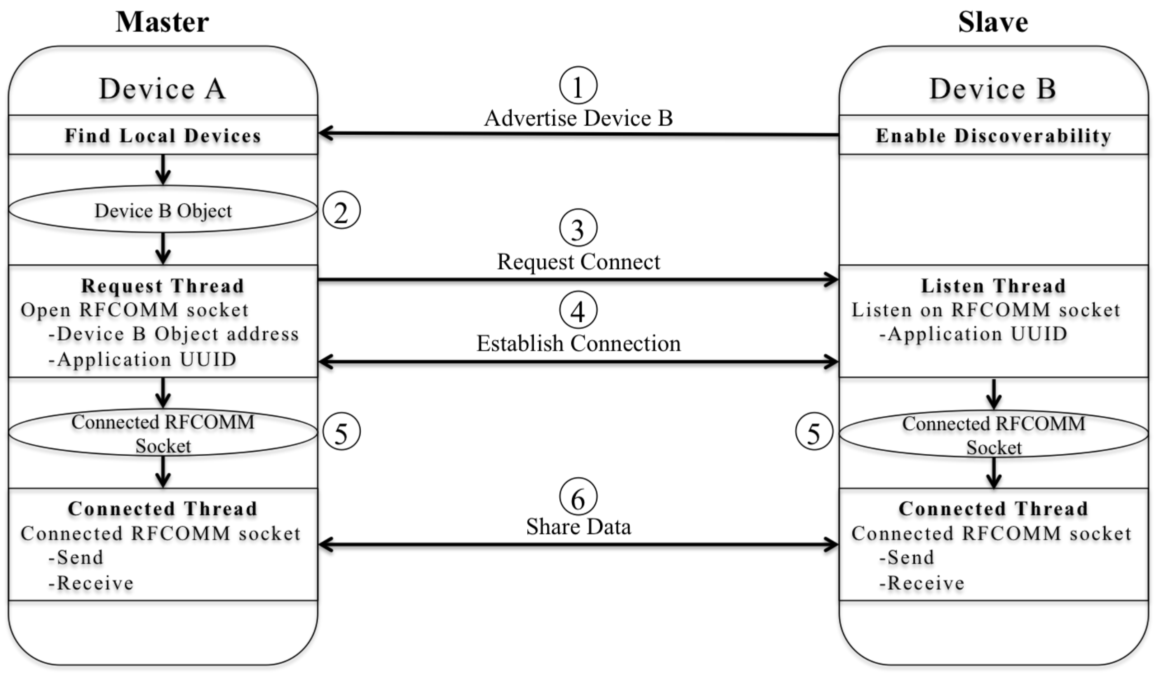


Figure 3.7: Interaction of threads between two devices

Connection Failure

This section refers to a Bluetooth network of devices as a scatternet, although the term is interchangeable with the piconet term. Due to the constantly changing scatternet topology, created while warfighters are on the move, the CM must handle inter-device connection failures; furthermore, it should attempt to reestablish connectivity with a device that has lost connectivity with the scatternet, and do so without unnecessarily draining the power supply of an individual device or the overall scatternet. In order to do this, the CM has a series of methods that attempt reconnection when an initial connection attempt fails as well as when an established connection between two devices fails.

Master Device Role When a user selects devices from the device selection screen, he or she expects connectivity with each device; however, failures do happen, and the system must be prepared for that situation. In the event that the master device's CM cannot establish an initial connection with a particular slave device, or if an established connection fails, the master will wait 10 seconds before making a second connection attempt. It will continue in this manner for a total of five reconnection attempts. On the fifth and final reconnection attempt, the CM creates a Reconnect Thread, which establishes a reconnection attempt timeline (details of the Reconnect Thread are discussed in a paragraph below, titled the same). In order to limit the number of reconnection attempts made immediately after an established connection failure, the master will solely attempt to reconnect during the first minute. This will allow the master to maintain authority over the piconet should it reestablish the connection, limiting extra communications at lower layer protocols. Figure 3.8 depicts the reconnection timeline of the master device.

Also occurring after an established connection failure, the master device determines if any other devices were reachable through that connection. After the fifth failed reconnection attempt, the master device will transmit a message to all other connected devices in the scatternet that the reachability to the slave device, along with the reachability to any other affected devices, is down. The master device will then create Reconnect Threads for each device that it has determined is no longer reachable.

Slave Device Role A limitation of the connection process is that the potential slave device does not know when another device is attempting to create an initial connection; therefore, a slave device has no way to recover from an initial connection attempt failure. After an established connection failure, however, the slave can act to reestablish a connection. When the failure occurs, the slave device creates a Reconnect Thread, which it will use to create a reconnection attempt timeline; furthermore, after waiting 60 seconds to allow the master device to reestablish connectivity, it will send a message to all other connected devices, notifying them of the failure so that they may take any necessary action. The slave device also determines that all other nodes which were reachable through that connection are now no longer reachable; therefore, it will have to attempt to reestablish a connection with them so that the scatternet can continue to operate. It will again create a Reconnect Thread for

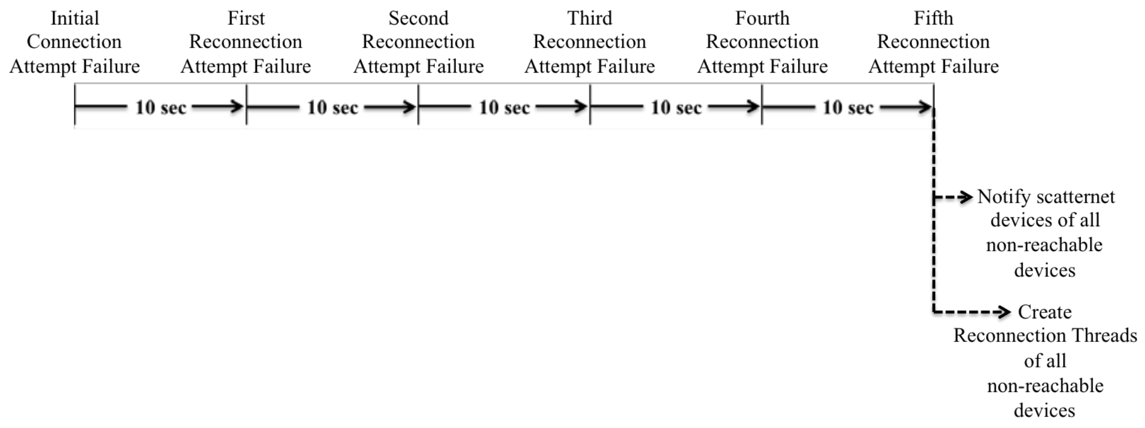

Initial Connection Attempt Failure
First Reconnection Attempt Failure
Second Reconnection Attempt Failure
Third Reconnection Attempt Failure
Fourth Reconnection Attempt Failure
Fifth Reconnection Attempt Failure
10 sec
10 sec
10 sec
10 sec
10 sec
Notify scatternet devices of all non-reachable devices
Create Reconnection Threads of all non-reachable devices

Figure 3.8: Reconnection attempt timeline after initial connection attempt failure

each device that was previously reachable through that failed link, it will also notify the rest of the remaining piconet/scatternet of the loss in reachability to those devices through that slave device.

Other Scatternet Device Role All other devices within the scatternet may rely on the disseminated information of the two devices that suffered the connection failure. When receiving a message about a connection failure, and what devices were lost due to the failure, the other devices in the scatternet must determine if those devices are no longer reachable through any other path. If they are not, then each scatternet device will create a Reconnect Thread for each device that is no longer reachable. The scatternet devices will continue to disseminate the connection loss message throughout the scatternet to ensure that all reachable devices receive it and can determine if they need to attempt a reconnection.

Successful Reconnection If the master device successfully reconnects to the slave device during one of the first five reconnection attempts, it will do nothing other than stop attempting any further reconnections. The slave device, upon recognizing that the master device has successfully reestablished connectivity, will stop its Reconnect Thread from attempting a connection; furthermore, it will not pass on the connection failure to any of the other devices.

Should the master device fail to reconnect during the initial five attempts, then all other devices which have created a ReconnectionThread will have the goal to reestablish connectivity. Once a device successfully establishes connectivity with any one of the lost devices, it will notify all connected devices of the status change. It will then terminate the Reconnect Thread for that device, as will all devices that received the successful reconnection status change. This will continue until all devices have reestablished connectivity, either directly or through another device.

Reconnect Thread When the Reconnect Thread is instantiated, it will determine the initial sleep time before it attempts a reconnection with the supplied disconnected device. It determines this sleep time through a random number generator. It generates a random time between 1 minute and the $n+1$ minutes, where n is the known size of the scatternet, counting in 6 second intervals. For example, if there are four devices in the scatternet, then the Reconnect Thread will select a time from the set of times 1:00, 1:06, 1:12, ... , 4:48, 4:54, 5:00. The time window size grows with the scatternet size to allocate more time between reconnection attempts in the event that all devices within the scatternet attempt to reconnect with a lost device.

The original implementation only chose an initial sleep interval from the same range, but at one-minute intervals. This led to a high probability that two devices would awake and then attempt to reconnect at the exact same time. This raised issues if both devices were the original master and slave because their simultaneous reconnection attempts conflicted over which device was the new master and which was the new slave, causing the application to crash and restart. With the devices having 10 times as many sleep times from which to choose, the probability of the original master and slave awaking at the same time decreases. In the worst case scenario, where they are the only two nodes in the network, there is a five-percent probability that both devices choose the same six-second time slot in a two minute window.

After the initial sleep and reconnect, the Reconnect Thread will then sleep for $n+1$ minutes before attempting the second reconnection. It will conduct the same pattern again, but after a third reconnect failure it will double the sleep time. It will continue to do so until doubling its sleep time meets or exceeds 20 minutes. It will continue to sleep at 20 minute intervals, until the device reconnects successfully or until it is notified of another devices successful connection.

Figure 3.9 depicts a Reconnect Thread's timeline, if it generates an initial sleep time of one minute, ten seconds, and it knows of four devices in the network (including itself). If the relative initialization time is 0:00 (minutes:seconds), then it will attempt its first reconnect at time 1:10. It will continue down the timeline if the reconnect attempts are unsuccessful.

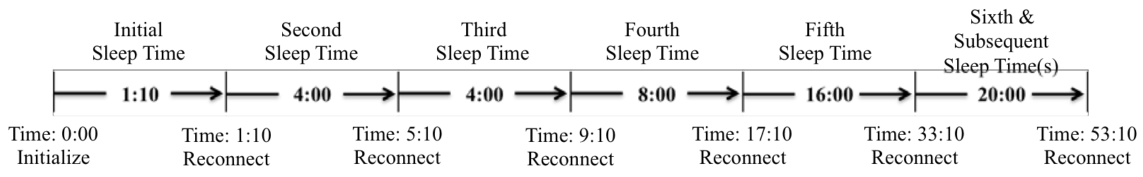


Figure 3.9: Reconnect Thread timeline with a known network size of four devices

Message Processing
Upon successful receipt of a CBMessage, the CM must properly process the message. In order to do so, it initially parses each segment of the CBMessage header and the payload. This section will discuss each phase of processing and how the CM processes the standard chat message and the connection status message. Chapter 4 will explain how the CM processes the test messages.

Sender The Connection Manger will determine if it knows the sender. If it is unaware of the sender, then it will build a database record of all required routing and connectivity status information for that device. The CM may be unaware of a distant scatternet device, so, in order to prevent a database crash, this check must occur.

Time Stamp Once the CM knows the sender, it will determine if it has previously seen this message. The CM keeps a mapping of known devices in the scatternet to the most current CBMessage time stamp from that particular device. If the CM processes a message with a time stamp that is older than the current mapping value, it dismisses the message as stale and drops it, neither saving the time stamp nor forwarding the message; otherwise, it determines it to be a new message, maps the message's new time stamp to the sender, and forwards the message on to all other connected devices.

Chat Message The CM must then process the message based on message type. The easiest message to process is the standard chat message sent from a user. In the case of a chat message, it will write the entire message, to include the header information and a timestamp of when the message was processed, to non-volatile memory in human-readable format. It will then display the message sender and the message payload for the user in the Chat Screen (see Figure 3.4 for example).

Connection Status Message The connection status message is potentially the most difficulty to process as the payload holds status information about varying scatternet devices, and the CM must verify that the status information applies to the host device. The CM first separates the payload into the different status updates (Note: A sender will only create and send a connection status message pertaining to the status of devices with which it is directly connected; however, it will forward all connection status messages from other devices within the scatternet, as long as it is not considered a stale message). The CM will then parse the three elements from each status update: the device name, the device MAC address, and the status (i.e., up, down, unknown). It will then verify if it is aware of the device to which the status pertains. If it is unaware of the particular device, it will create all of the necessary database information for it.

If the connection status informs the CM that the device connection is established (i.e., up), it will add the device to the routing table with the next hop being the directly connected device from which the message came. It will also check if there is a Reconnect Thread running for that device, and if so, it will cancel it. Should the status inform of device where the connection is down or unknown, the CM will remove the device from the routing table. It will also create a Reconnect Thread for this device, ensuring the resiliency of the scatternet.

3.6 Summary

This chapter discussed the inner-workings of the ChatterBox application. It discussed the initial operational concept for the application development: to enable small warfighting units with direct communications using only the embedded Bluetooth radio of a tactical smart device. The chapter then walked through the three application screens with which the user will interact, and some of the subcomponents of each screen. Lastly, it discussed the CM: the core of the application. The CM creates connections, distributes data across connections, processes messages, and most importantly builds a smart scatternet that is capable of reconnecting disconnected devices regardless of where they emerge on the scatternet topology. Chapter 4 will detail testing the viability of the system in operation.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 4: Testing and Evaluation

4.1 Overview of Testing

The evaluation for this thesis involved testing three different aspects of the ChatterBox network: throughput at the application, order of message delivery at the user application layer, and network recovery from connection loss. The throughput test measures the amount of data transmitted and processed by the user application per unit of time. The second test, testing message delivery order, verifies that messages received across the system do so in the order in which they were sent, without loss. The last test, network recovery from connection loss, examines the capability of the ChatterBox Bluetooth scatternet to recover from the loss of a master or slave device (i.e., rebuild the broken connections of those devices within connectivity range). These tests demonstrate the feasibility of using a Bluetooth network as the base layer for the greater infrastructure-less communications network of tactical smart devices, as represented in Figure 3.1.

4.1.1 Devices

The throughput test uses four different Bluetooth capable devices: one tablet and three smartphones. Table 4.1 shows the basic information of each device. The test distinguishes between the devices based on their assigned device names (i.e., Alpha, Bravo, Charlie, Delta). Samsung manufactures each of the phones and all phones run the Android operating system. The devices all use Bluetooth 3.0 interface cards developed by Samsung.

The message order and network recovery tests use the above four devices, plus an additional four to evaluate characteristics of a more robust scatternet made of multiple piconets.

Device Name	Device Model	Device Type	Bluetooth Version	Android Version
Alpha	Samsung Galaxy Tab 10.1	Tablet	3.0	3.0
Bravo	Samsung Galaxy Nexus	Smartphone	3.0	4.3
Charlie	Samsung Galaxy Nexus	Smartphone	3.0	4.3
Delta	Samsung Galaxy Nexus	Smartphone	3.0	4.3

Table 4.1: Test-device information

Samsung manufactured the additional devices, which all run the Android operating system. Table 4.2 provides the basic information for each device.

4.2 Throughput Testing

Testing the throughput of the Bluetooth, text-based, messaging application requires four fundamental steps: throughput test development, baseline testing, field testing, and data parsing. Baseline testing provides a best-case scenario (i.e., minimum distance) for throughput of the application; whereas, the field testing first determines the maximum distance at which two devices can maintain a Bluetooth connection and then tests the throughput at that distance. The data parsing portion of the test requires pulling the data size and the timing information from each device and determining the average throughput for each connection.

4.2.1 Test Plan

The throughput test implements a series of seven test phases between each device-pair connection (e.g., Alpha-Bravo, Alpha-Charlie, etc.). Each phase builds a set of fixed-sized messages—based on payload size—and continuously sends that message across the Bluetooth connection 100 times. The payload sizes start at 128 bytes and double at each consecutive phase, up to 8192-byte payloads.

During initial test runs, limitations on the receiving node's message processing software were discovered: the messages came in at such rapid pace that the receiving devices' message processor, after delivery from the RFCOMM socket buffer, would sometimes include one or more messages as a payload of the first message in the buffer. This meant that the messages required a start-of-message and end-of-message delimiter—one byte for each—at the application level. The final calculations take into consideration each of the messages

Device Name	Device Model	Device Type	Bluetooth Version	
Echo	Samsung Galaxy Tab 10.1	Tablet	3.0	3.0
Fox	Samsung Galaxy S III	Smartphone	4.0	4.0
Golf	Samsung Galaxy S III	Smartphone	4.0	4.0
Hotel	Samsung Galaxy Nexus S	Smartphone	2.1+EDR	2.3

Table 4.2: Additional test-device information

exact payload size along with the varying header size and the two-byte message delimiters.

The receiving device saves the message size and receipt time, using nanosecond precision, into an array. It does not save the message. The actual messages hold no pertinent information and saving them would unnecessarily consume limited memory resources. Upon completion of each of the seven test phases, the receiving device uses a separate thread to write the array of messages to non-volatile memory. Waiting until the end of each phase to save the data minimizes the time between message timestamps, and it frees memory resources for subsequent phases.

The user activates the test procedure by selecting it from a dropdown menu. Both devices display the progress of the test as chat messages, notifying the user at the start of the test, the start of each phase, and the completion of the test. In order to gain more data points for analysis, when the user activates the test, the test runs ten consecutive times.

4.2.2 Baseline Testing

This research requires a baseline test in order to determine the baseline throughput that ChatterBox could achieve at the minimum distance (< one foot) on a flat surface—two identical wooden platforms. In order to build such a baseline, each device individually connects with every other device and then runs the test. In an attempt to minimize conflicting signal noise in the 2.4 to 2.485 GHz spectrum, the tests were conducted at a remote location—an isolated football field (see Figure 4.1 for test location and examples of the wooden test platforms)—where no discoverable Bluetooth devices or active WiFi signals were detected. While the tests were conducted, all other devices were powered off, except for the two test devices.

For the baseline tests, each pair of test devices sat parallel to each other, in a portrait orientation, exactly ten inches apart at the inner edges. Four separate devices create 6 different bi-directional connections for a total of 12 testable links. At the completion of each test, the Bluetooth connection and the ChatterBox application on each device were terminated, and both devices were powered off. This added time to the baseline testing; however, it gave the phone, application, and connection a common starting point. At the completion of all 12 tests, the saved test files were downloaded through a USB connection.

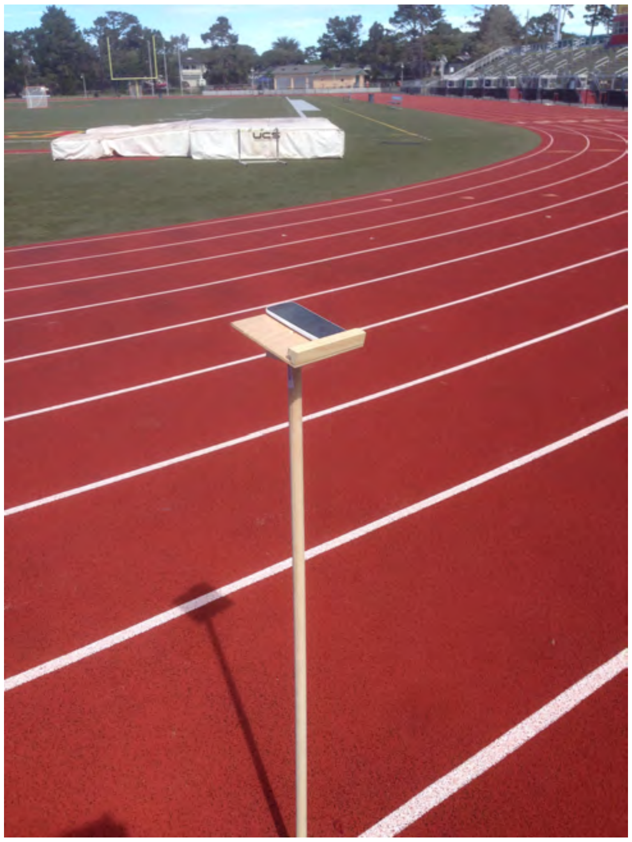


Figure 4.1: Throughput testing location and device platform

4.2.3 Field Testing

The field testing consisted of the same manner of tests as the baseline; however, the devices sat at a far greater distance from one another, a distance that was determined prior to conducting any tests.

The maximum connection distance for the field tests came from pairing the devices through a Bluetooth connection. One device sat at the end of the football field, while the tester held the other, walking a straight path along the football field, monitoring the connection. When the phone notified the tester of the connection break, the distance, in meters, was noted. The tester than walked back toward the stationary phone until the connection reestablished, making note of the reconnection distance.

The first pair of devices to test for connection distance, a tablet and phone, far surpassed the commonly advertised Bluetooth connection distance of 40 yards. The test terminated at 160 yards only due to the physical confines of the football stadium. At that distance, the tablet and smartphone continued to hold a solid connection. This held true for all tablet-to-smartphone connections; therefore, all tablet-to-smartphone, and vice-versa, throughput tests were conducted at 160 yards.

Since the phones could maintain a two-way connection with the tablet at 160 yards, it seemed that they should maintain the same connection distance for phone-to-phone; however, this did not hold true. This phenomena is most likely due to the larger antenna gain of the tablet. The phone-to-phone connection achieved a maximum distance of 120 yards, but the connection would break within the first few trials of the throughput test. The maximum sustainable phone-to-phone connection stabilized at 60 yards. The phone-to-phone throughput tests all ran at this distance.

4.2.4 Data Processing

The last step in the test process requires processing the collected data to determine the average throughput for each phase of each test trial. A Python script was created to parse the data collected from each test and process it into useable details. The calculated average-throughput results of the script were entered into Excel©for graphing purposes.

The Python script has three modules: The first module separates the results from each test trial into the corresponding seven phases, based on payload size (e.g., 128-bit payload, 256-bit payload, etc.). It then determines the per-message total byte count, the receipt time of the first message, the receipt time of the last message, and the total number of messages transmitted for that phase. It multiplies the message size by the total number of messages to determine the total number of bits received during that phases time span. Finding the time difference between the first and the last messages, the program divides the total bits received by that time difference to calculate the throughput for that phase. A sample test file was first created and run to verify the calculations of this module.

The second module takes the calculated throughput at each phase of the 12 test trials and calculates an average throughput for that phase of testing across all test trials. The last module combines each of the phase's average throughputs across the connection into a single file for graphing and analysis.

4.2.5 Throughput Results

The results of the baseline and field tests are divided into four graphs. The separation comes from the distinct difference in connection distances between the tablet-to-phone connections (160 yards) and the phone-to-phone connections (60 yards); therefore, this graphical separation allows easy comparison of the baseline throughputs to the corresponding distance throughputs.

Figure 4.2 shows the throughput results of the tablet-to-phone connection baseline and 160-yard throughput tests. The 8192-byte payload messages achieve the maximum baseline throughput of 1.857 Mbps between all connections . In comparison to the 160-yard tests, the 8192-byte payload messages achieve an on-average throughput of 360.4 Kbps—over a five-fold decrease in performance. The difference in performance is attributed to the distance between devices.

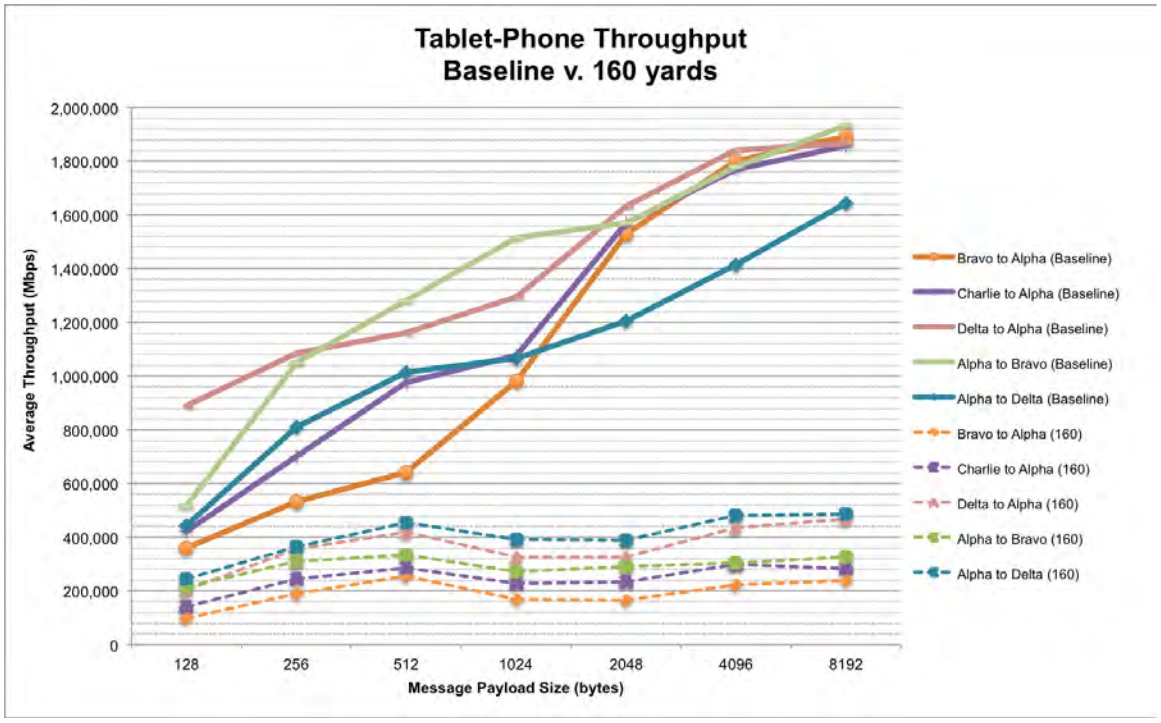


Figure 4.2: Tablet-to-phone connection throughputs. Baseline results are annotated by solid lines, and 160-yard results are annotated by dashed lines.

Figure 4.3 depicts the phone-to-phone baseline and 60-yard throughput test results. The highest throughput for the baseline tests came, again, from the 8192-byte payload messages, averaging 1.875 Mbps for all connections. The common trend did not hold for the 60-yard connections; instead, the highest throughput came from the 4096-byte payload messages, which achieved an on-average throughput of 698.4 Kbps. At 60 yards the connections can transmit one-third the amount of data than that of the baseline, in the same time span.

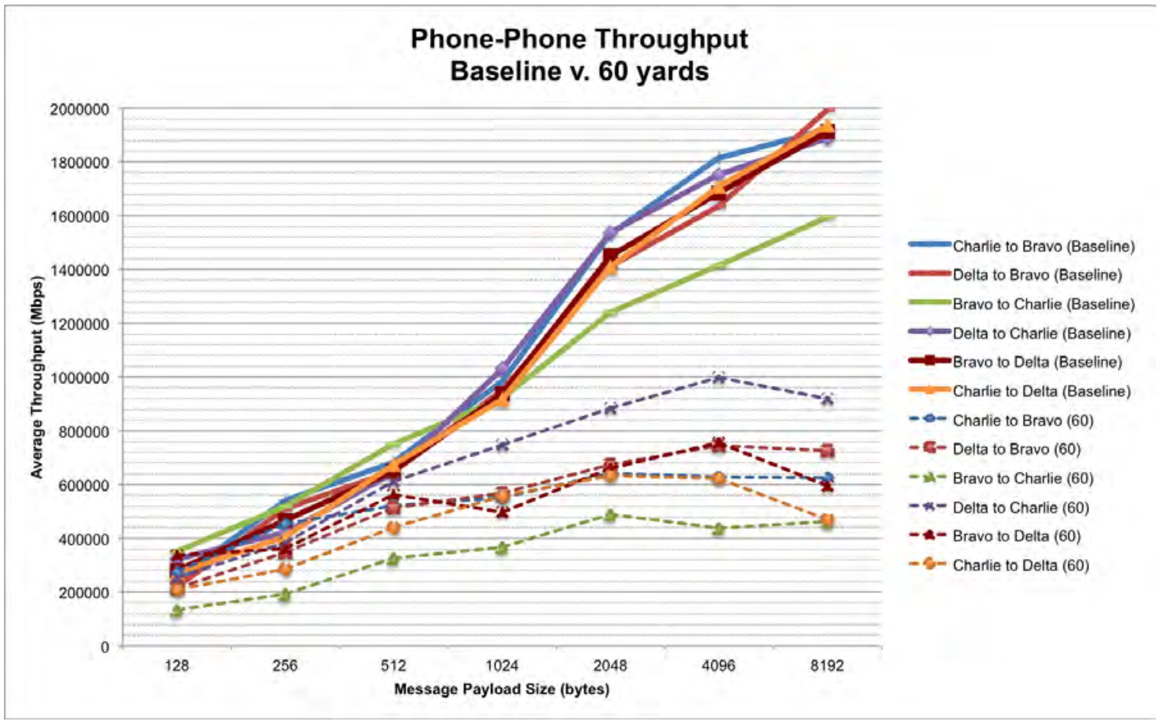


Figure 4.3: Phone-to-phone connection throughputs. Baseline results are annotated by solid lines, and 60-yard results are annotated by dashed lines.

4.3 Message Order Testing

The Bluetooth core protocols provide reliable delivery of data at their respective layers; however, the Bluetooth specification states that the core protocols do not offer the functionality of network routing. As such, the forwarding of messages from one device to the next happens at the user-level application, in the case of ChatterBox, the CM performs this function. Therefore, tests must be conducted to ensure that the forwarding process happens in a manner that does not interfere with the order of message delivery. The CM has a built-in test that transmits a set number of messages of increasing size throughout the network. When the user activates this test, theses messages propagate across the scatternet, at the completion of the test, all receiving devices will report the number of messages received in order, verifying that all messages were delivered in order. This section will discuss the details of the test software, the test scenarios, and the results of the tests.

4.3.1 Test Software

The message ordering test can be initiated through the chat screen menu, detailed in Section 3.5.2. Selection of the "Order Testing" menu item will initiate a thread, named the OrderingTestThread, within the CM. The OrderingTestThread of the test device—the device initiating the test—sends a message across the scatternet that tells the other devices that the message ordering test has been initiated. Upon receipt of this message, the receiving devices prepare the counters necessary to determine the number of received test messages. The test device then sends 6 rounds of test messages; each round includes 50 messages of increasing payload size. The first round has a message payload size of 128 bytes, then each subsequent round doubles the payload size of the previous round, up to 4096 bytes. The test device announces the start of each new round by sending a message stating the number of messages and payload-size of each message in the round. At the conclusion of all six rounds, the test device sends a message stating the conclusion of test. This final message triggers the receiving devices to display, through system message on the chat screen, the number of messages received.

The message logs, saved to non-volatile memory, are downloaded and processed to verify that the message time stamps are in ascending order

Limitations

The largest limitation on the test is the payload size. As shown in the throughput test results, the ideal payload size is 8192 bytes; however, during development of the message ordering test, the test device suffered from memory allocation issues which prevented sending a series of different 8192-byte, or higher, payload messages. The throughput test uses the same message and sends it repeatedly, which does not invoke the same memory allocation issues. Many steps were taken to reduce the amount of memory used, but the best case scenario for the test involved 6 rounds of 50 messages at the increasing payload size.

4.3.2 Test Scenarios

Testing the message delivery order of the routing system involves analyzing both piconets and scatternets of varying size and complexity. The initial tests involved a simple piconet of two devices, initiating the test from both devices. The piconet testing grew in size and complexity, testing an eight-device piconet (one master, seven slaves), initiating the test from three different slave devices and the master device. The last test involved building a three-piconet scatternet where two piconets contain three devices and one piconet contains four (the masters of the two smaller piconets serve as slaves in the larger). The final test scenario initiated tests from a slave in each piconet, one master-slave device, and the one master device. By initiating the test from multiple vantage points within each scenario, it ensures that the master, slave, and master-slave roles all perform the forwarding and processing of messages in the same manner. Figure 4.4 depicts the network topology of each test scenario with the test initiators identified for each scenario.

Although inter-device distance played an important role in the throughput results, it was not considered an issue for this test because the layer being tested was that of the application. The Bluetooth core protocol stack provides a QoS that includes guaranteed delivery, so the focus of the test was on the application's ability to process messages and forward them along the other logical links in the network without change to order or loss. Since this processing and forwarding component operates independently of connection distance, and because lower-layer Bluetooth protocols offer guaranteed delivery, testing at varying distances was deemed inconsequential. Of note, all tests were conducted at a distance of less than one meter.

4.3.3 Test Results

Each scenario was created and tested through three separate trials, varying the roles of the devices within the network, to ensure reproducibility of test results. In the first trial of the first test scenario (a piconet of two devices), a tablet served as the master to a smart phone; in the second trial, a tablet served as a slave to a smart phone; and in the last trial, a phone was tested with another phone. In the second scenario (a piconet of eight devices), the first trial involved one tablet as master to one slave tablet and six slave phones; the second trial involved one phone serving as master to two slave tablets and five slave phones; and the last trial involved the other tablet serving as master to the other seven devices. The three trials

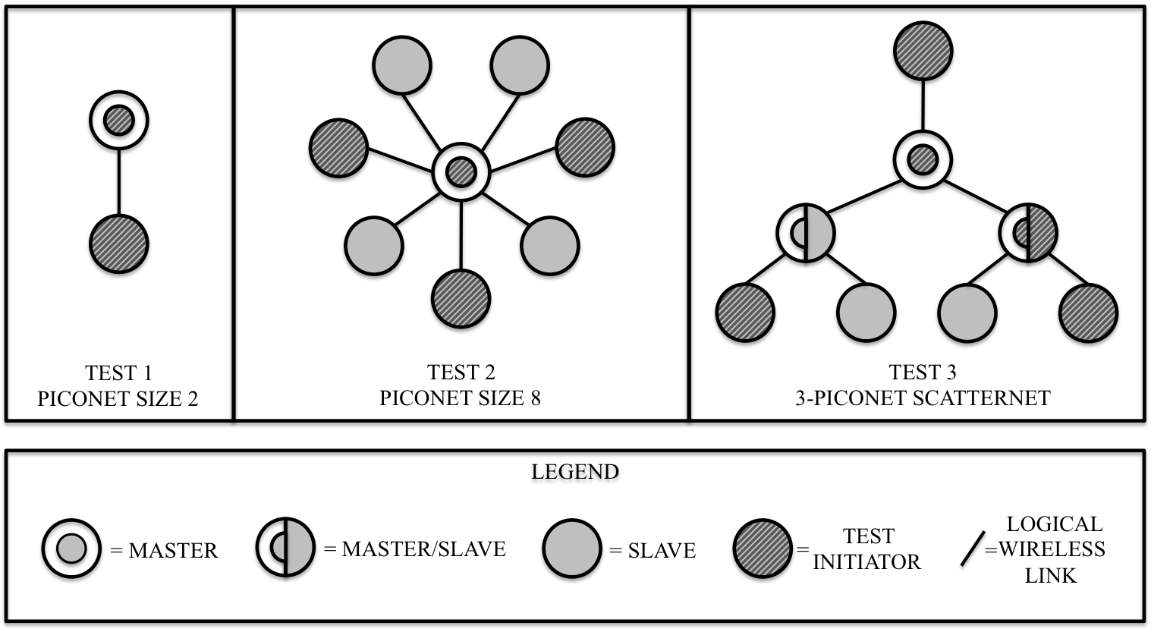


Figure 4.4: Graphical representation of message ordering test scenarios. Dark, solid-filled devices identify those devices used to initiate the test software.

of the last test scenario involved a tablet as master with a master-slave tablet and phone, a master phone with a master-slave tablet and phone, and a master phone with master-slave phones.

Each trial of tests included initiating the tests from the designated devices, identified in Figure 4.4. Upon completion of each trial, the message logs of each device were analyzed to verify the ascending order of time stamps. In all trials of all test scenarios, each receiving device reported receiving 300 test messages, and log verification ensured that all messages were received and processed in ascending order. Table 4.3 outlines the test scenarios, the device topology for each trial, the number of messages reported received by each device (300 messages for each initiating device), and the percentage of messages delivered in order as verified by device log timestamps.

4.4 Network Recovery from Connection Loss

Warfighters will inevitably move about the battlefield, and they will not always be within connectivity range of one another nor will the topology of their networked smart devices remain the same. The ChatterBox network must, therefore, gracefully handle loss of a

Test Scenario	Iteration	Master	Slave	Number of Initiating Devices	Total Messages Received by Each Network Device	Percent Messages Verified Delivered in Order
1	1	Tablet (Alpha)	Phone (Bravo)	2	600	100%
	2	Phone (Fox)	Tablet (Echo)	2	600	100%
	3	Phone (Charlie)	Phone (Hotel)	2	600	100%
2	1	Tablet (Echo)	Tablet (Alpha) Phones (B-D, F-H)	4	1200	100%
	2	Phone (Golf)	Tablets (A, E) Phones (B-D, F, H)	4	1200	100%
	3	Tablet (Alpha)	Tablet (Alpha) Phones (B-D, F-H)	4	1200	100%
3	1	Tablet (Alpha) Master-Slaves (E, F)	Phones (B-D, G, H)	5	1500	100%
	2	Tablet (Delta) Master-Slaves (E, F)	Tablet (A) Phones (B, C, G, H)	5	1500	100%
	3	Phone (Bravo) Master-Slaves (C, D)		5	1500	100%

Table 4.3: Breakdown of message ordering test scenarios. Devices are listed by common names, or abreviations thereof. Full device information can be found in Tables 4.1 and 4.2

device from the network and a change in network topology. This should be done in a timely and seamless manner.

The last, and potentially most important, test conducted on the ChatterBox network is that of its resiliency to connection loss, and its reconnection capability. This test examines two aspects of the network resiliency: the time taken to reconnect a lost device and the time taken to reconfigure the network to continue communications with all devices within Bluetooth range. This section will discuss the testing framework and the results of the tests.

4.4.1 Testing Framework

The framework for this test is broken into three scenarios, based on original network topology. It is imperative to emphasize the point that only the original topology is well known and understood. As the ChatterBox network recovers from loss, it rebuilds itself into a topology that may or may not be the exact same. The point of the test is not to verify that the topology is the exact same, rather to verify that the topology created still allows communication of all devices within range.

The first test scenario involves a piconet of only two devices. While this scenario has limitations in the ability to test the networks ability to autonomously reconfigure itself (because there are not enough test devices to do so), it will verify that the ReconnectThread attempts reconnections in the desired timeline, and that two devices can reconnect, regardless of the length of time they are separated.

The second scenario, involving a piconet of eight devices, is broken into two sub-scenarios. The first sub-scenario examines the effects of removing the master device from the network, measuring the length of time for the other seven devices to rebuild the network, and the length of time to reincorporate the removed master device, once moved into connection range. The second sub-scenario examines the effects of removing the master and three slaves from the piconet. This test will measure the ability of both smaller piconets to recover and then reintegrate once brought within connectivity range.

The last scenario verifies the ability of a scatternet to recover from loss when the master device, a master-slave device, and two slave devices are removed from the scatternet. Again, the test will measure the time taken for both sub-networks to rebuild and then reintegrate when brought back within range.

Figure 4.5 depicts the three test scenarios, including the second scenario's sub-tests. The dark shaded nodes identify those devices which were removed from connectivity range of the lighter devices.

4.4.2 Test Results

During development and implementation, the first scenario was conducted numerous times across all devices. For the final evaluation, two smart phones (devices Bravo and Charlie)

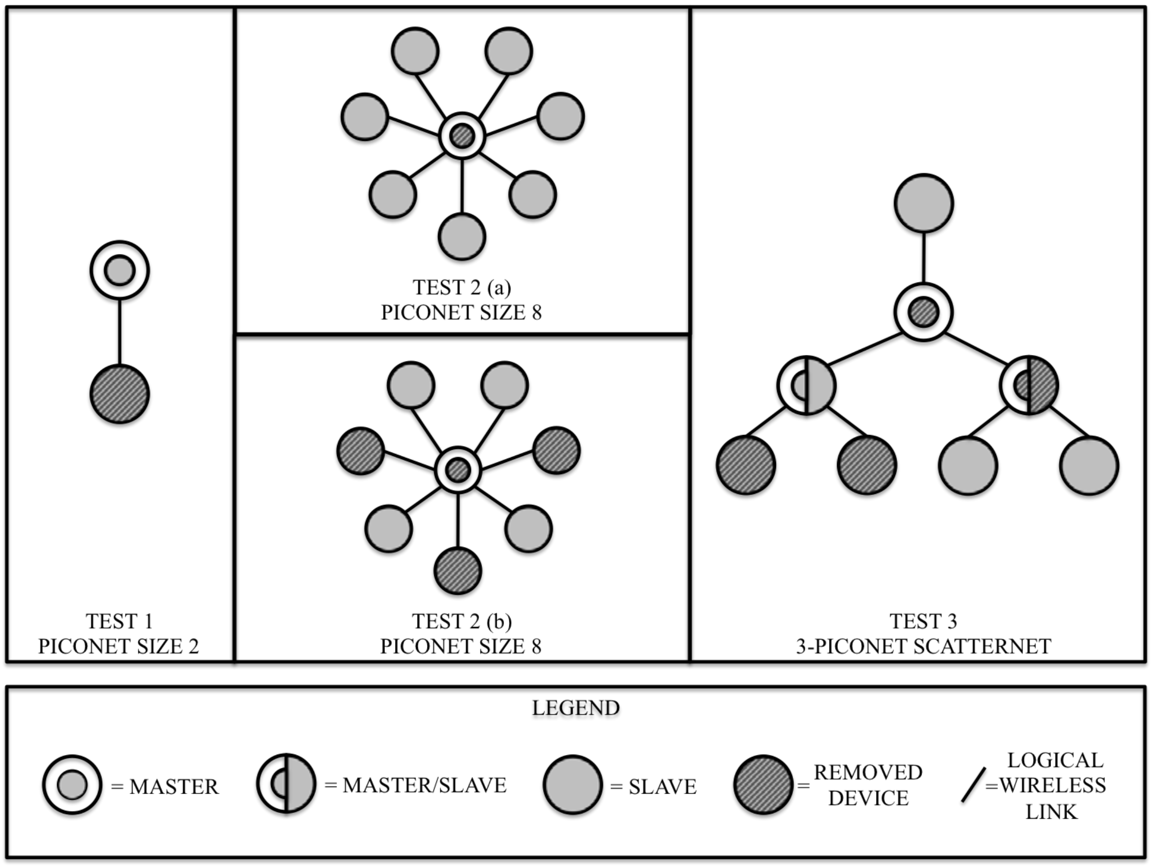


Figure 4.5: Graphical representation of network recovery test scenarios. Darkened devices identify those devices removed from initial network topology.

were used since smart phones have a shorter connection distance and can therefore more rapidly create a connection loss. The final test was conducted 20 times, measuring the time difference between the notification of connection loss and the notification of connection reestablishment. The average time for reconnection was 41.87 seconds with a standard deviation of 13.22 seconds. There were five occasions where the time to reconnect exceeded one minute. Of those five, three of them resulted in a role-reversal between the master and slave, and one resulted in the master device application crashing, as both devices attempted to reconnect near simultaneously. In the occasion where the device application crashed, when the application was restarted, the two devices connected without user involvement in less than one minute. Of note during testing, the two devices were left outside of connectivity range for over six hours. When the devices were brought back within connectivity range, they reconnected in under seven minutes. Although this amount of time to wait for a connection is quite large, the ReconnectThreads of each device after 6 hours of separation only attempt a reconnection every 20 minutes; furthermore, with only two devices in the network, the worst-case wait time for reconnection could approach 20 minutes. Delaying the time between reconnection attempts prevents unnecessarily attempting reconnections, and therefore wasting battery power, more frequently. Further work could include optimizations to this process.

Building the exact topologies laid out in second and third scenarios proved difficult because of the networks ability to reconfigure quickly. If a connection failed during the setup process, the network would begin to take over and reconfigure itself as needed. In both cases, the test topologies were built as designed three times without interruption.

In the case of the second scenario, sub-scenario one, the average time taken for the seven devices to reconfigure the network topology so that all seven devices communicated took 93 seconds. With all seven devices connected, the network reintegrated the removed device in under 30 seconds. Sub-scenario two had similar results, although the subnetwork of four devices that did not include the original master took slightly longer to reconfigure, taking an average of 112 seconds to rebuild. Reintegration of both four-device piconets required less than 30 seconds in all tests. In the final scenario, the two four-device topologies reconfigured themselves in 84 seconds, on average, and reintegration took 37 seconds.

4.5 Summary and Conclusion

Throughput Conclusion The tests offer some expected and very unexpected outcomes of Bluetooth technology when linking mobile devices to send and receive data. As hypothesized, the amount of throughput has a directly inverse relation to the distance between devices: as the inter-device distance increases, the throughput decreases. This is most likely due to data transmission errors at the lower layers of the Bluetooth protocol stack, which then require retransmission of data at these lower layers. When the data is then properly transmitted and brought up the protocol stack, the user application sees this as a decrease in throughput. Hardware that can analyze Bluetooth transmissions would be required to further analyze this hypothesis, but it exceeds the scope of this thesis. The throughput also shows the common tendency to increase as the payload size increases, up to a given point;

however, the 160-yard throughput does have a decline in throughput from the 512-byte payload through the 2048-byte payload. This may result from inefficiencies in the software that hit a limitation at those message sizes, but the throughput increases again at the 4096- and 8192-byte payload sizes.

The very surprising outcome from the tests came from the achievable connection distances of the devices: the smaller 60-yard communication distance still outperforms the advertised Bluetooth distance of 10 to 40 yards. The 160-yard communication distance comes as a far greater surprise—a four-fold increase over the advertised distance. The most likely reason for this is that the tablet has a higher gain antenna that can allow greater distance connections. A search for information regarding these antenna only produces the manufacturer and version number.

These achievable distances, paired with their measured throughputs, show that a smart device network built on a Bluetooth foundation can exist as a viable option for text-based communications systems on the battlefield; however, further testing would be required to determine the effects of terrain and obstacles on the network.

Message Order Delivery Conclusion This test demonstrated the reliable data delivery of the underlying Bluetooth protocol stack as much as it does the user application. Since Bluetooth guarantees delivery with an established connection, as long as the user application handles messages in the order in which they are received from the Bluetooth stack through the RFCOMM socket, then the processing and forwarding should continue in an ordered fashion. The tests verified that the interaction of multiple threads within the user application did not disturb the ordering of messages from the lower layers. Although the results are unsurprising, the test was a necessary step in verification of the overall system.

Network Recovery Conclusion The test results show that a small network (eight devices or fewer) can reconfigure itself when a network loss occurs. Waiting nearly two minutes for the network to reconfigure may or may not be a desirable tradeoff for the warfighter, but further operational, human testing would be required to understand the implications of such a delay on the application's utility for the warfighter.

The most notable observation from the last two scenarios of this test is that the network has an elastic quality where devices swap connections unnecessarily. As a simplified example, if Alpha is connected to Bravo, and Bravo to Charlie, the reconfiguration behavior may cause Alpha to connect with Charlie, thereby breaking the connection established with Bravo. This may go on for several iterations, swapping every two to three minutes; however, it does stabilize. It was also found that chat messages help stabilize this behavior because when a device receives a message from another, it notes that their exists a path to that device through its one-hop neighbor and does not attempt a reconnection.

Overall Conclusion These tests show that the embedded Bluetooth system within smart devices can be used to establish the base-layer of a useable, infrastructure-less communications network. The distance covered is shown to be far greater than that which is advertised. Although this distance is not great, as compared to larger, infrastructure-based networks, it permits communications between closely geo-located members of a tactical unit. The throughput at these distances, though not necessarily ideal for large data transfer, will provide adequate real-time chat capability. Since Bluetooth does not provide message forwarding, the user application ensures that the messages are processed and forwarded in the order in which they were received, guaranteeing that all tactical unit members receive communications in the same manner. Lastly, the ability of the network to handle device loss is imperative as unit members move about the battlefield. The final test shows that the network can reconfigure itself, into as many sub-networks as it requires to still maintain communications with part of the tactical unit. Although the ChatterBox system works, it is still in its infancy, and the next chapter will explain some recommended follow-on work which will create a more robust infrastructure-less network.

CHAPTER 5: Conclusions and Future Work

The aim of this thesis was to build an infrastructure-less, adaptable, real-time communications network of mobile devices through the development of user application software. The designed system gives users the ability to conduct real-time chat across a Bluetooth-based network. In our informal experiments, we observed that it provides a means of creating and managing a network, forwarding network traffic, and seamlessly reconfiguring the network topology in response to connection loss.

At the 160-yard, sustainable, line-of-sight connection distance, the application is capable of nearly 500 Kbps with an average throughput of 360 Kbps and a minimum throughput of 100 Kbps. At 60 yards, the application delivers a maximum, average, and minimum throughput of 1 Mbps, 700 Kbps, and 120 Kbps, respectively. This throughput, although not ideal for real-time voice or video, is more than adequate to conduct network management and real-time chat.

Although the results from the testing of ordered message delivery are underwhelming, producing the expected behavior without error, it shows that application level software can fill the gaps of lower layer protocols when creating an infrastructure-less network. Since application level software is capable of bridging the gap between multiple Bluetooth connections, it is conceivable that it can bridge the gap between multiple different types of connections (Bluetooth, Wi-Fi Direct, cellular, etc).

The developed network has proven the ability to adapt the drastic changes in network topology, including the loss of the central scatternet device that connects multiple piconets. The network is flexible enough to establish newer subnetworks of the original based, when the original network is unachievable due to device loss. It has also shown that when all devices are back within connection range, it can rebuild a network of all devices from the subnetworks. This flexibility does have limitation in the amount of time required to re-establish connection—anywhere from tens of seconds to minutes. This research builds the foundation for future research and enhancements to improve the timing and resource management—battery power—of the system.

This software builds the foundation for an advanced infrastructure-less network that will allow the warfighter to conduct C3, using the intuitive, lightweight, highly-mobile attributes of smart devices.

5.1 Future Work

This application provides the foundation for an infrastructure-less mobile device network; however, there are areas where expansion could increase the scalability of the network and enhance performance. Incorporating Wi-Fi Direct and cellular capability into the network will increase the network's geographic coverage area and the number of connectable devices. Incorporating multiple types of transmittable data, such as imagery, audio, and video, will improve the command and control (C2) performance of system. This section discusses these items in detail.

5.1.1 Wi-Fi Direct

The Bluetooth network proved to have considerable more coverage area in clear operating environments than is advertised by smart device manufacturers; however, Wi-Fi Direct is advertised to have a far greater coverage area and throughput than that of Bluetooth. Incorporating Wi-Fi Direct capability into the ConnectionManager module will allow far greater flexibility in managing the network. It is recommended that the ConnectionManager initiate connections through Bluetooth, and upon doing so, gathers all connection information about the networked devices, specifically Wi-Fi Direct MAC addresses. If the ConnectionManager requires greater throughput, or if a connection is lost and not quickly re-established with Bluetooth, then the ConnectionManager could create a connection using Wi-Fi Direct.

Running multiple wireless protocols can have negative consequences, primarily a more rapid depletion of battery power. The ConnectionManager should minimize the use of multiple wireless protocols to the maximum extent practicable. This does not necessarily mean that all connections should be primarily maintained with Bluetooth. Perhaps connecting multiple devices under a single Wi-Fi Direct umbrella would require less battery power than individual Bluetooth connections. More research is needed to understand the benefits of different wireless protocols in establishing, maintaining, and transmitting data over mobile network connections.

5.1.2 Cellular

The cellular capability of smart devices is typically associated with infrastructure-based networks; however, it is a wireless medium common to all smart devices that could be manipulated to support an infrastructure-less network. Future work could include altering of smart device cellular systems to create a network connection between devices. This may require transmitting data in covert methods over the cellular control channels. If possible, adding this capability could provide broader network coverage. Alternatively, Qualcomm Technologies has worked since 2013 on developing LTE Direct, a device-to-device connection service over regulated communications bands [16]. Tapping into this resource could expand, by orders of magnitude, the connectivity distance and throughput between devices. Working with this technology will require registration with Qualcomm Technologies, who can provide access to information, discussion forums, and LTE Direct APIs. Testing these improvements will require thorough legal coordination as it manipulates data transmitted over regulated communications bands.

5.1.3 Imagery

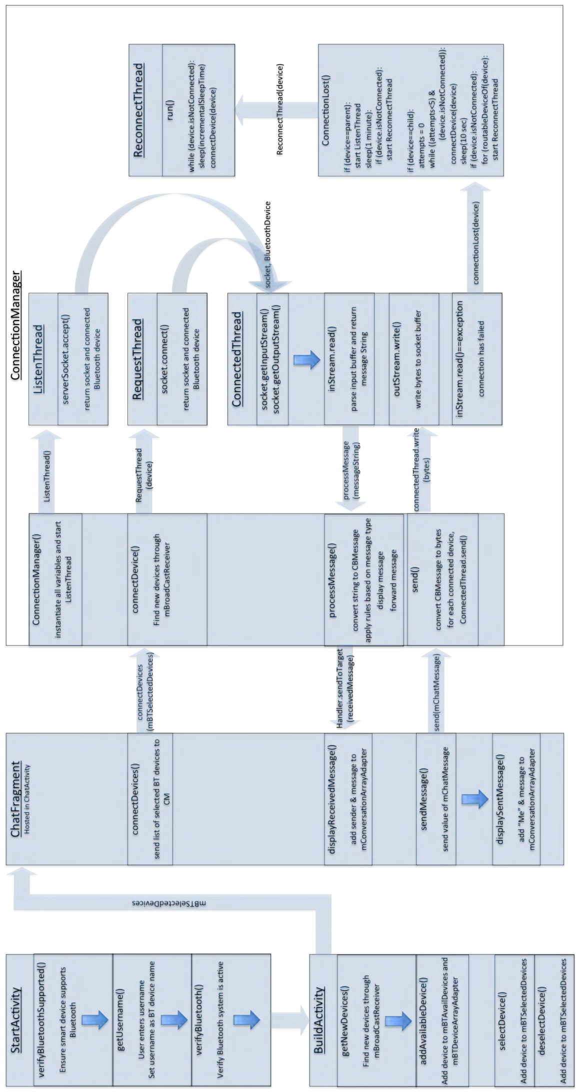
The ChatterBox application only supports transmitting text-based messages across the network. Incorporating the ability to transmit imagery would provide warfighters with the ability to share a common situational picture. This feature would require modification to the GUI. It would also require enhancements to the message processing capability of the ConnectionManager, so that it can proper display the images when received. The network may require Wi-Fi Direct or other wireless protocols to transmit an image, depending on the size of the image. Further investigation would be required to understand how certain image sizes affect the overall network.

5.1.4 Voice and Video

Voice communication would prove instrumental in C2 of warfighting units as it is the traditional fall-back for all tactical communications. As Bluetooth is often used to transmit audio from a smart device to a wireless speaker, it is conceivable that this enhancement would work for at least a single point-to-point connection. Future work should incorporate voice transmission across the network, where one user could speak—real time—to all participating users on the network. This would require analysis of how such data transmissions affect the ability to maintain a stable network, battery consumption of such features, how many devices can transmit real-time voice simultaneously, and which wireless communications protocols are more suited for such an enhancement. One potential solution may be the incorporation of voice-to-text at the sender and text-to-voice at the receiver allowing for leveraging the ChatterBox application prototyped by this thesis.

Along similar lines, enhancing the system with real-time video capability would add another level of common situational awareness amongst combat units. Future work could begin with sending recorded video across the network. Once the application can transmit pre-recorded video across the network, then work could advance toward streaming real-time video. Again, incorporating such enhancements would require research into suitability of certain wireless protocols, network behavior, numbers of simultaneous transmissions, and battery consumption.

APPENDIX: Source Code Diagram



StartActivity
verifyBluetoothSupported()
Ensure smart device supports Bluetooth
getUsername()
User enters username
Set username as BT device name
verifyBluetooth()
Verify Bluetooth system is active
BuildActivity
getNewDevices()
Find new devices through mBroadCastReceiver
addAvailableDevice()
Add device to mBTAvailDevices and mBTDeviceArrayAdapter
selectDevice()
Add device to mBTSelectedDevices
deselectDevice()
Add device to mBTSelectedDevices
mBTSelectedDevices
ChatFragment
Hosted in ChatActivity
connectDevices()
send list of selected BT devices to CM
displayReceivedMessage()
add sender & message to mConversationArrayAdapter
sendMessage()
send value of mChatMessage
displaySentMessage()
add "Me" & message to mConversationArrayAdapter
connectDevices (mBTSelectedDevices)
Handler.sendToTarget (receivedMessage)
send(mChatMessage)
ConnectionManager
ConnectionManager()
instantiate all variables and start ListenThread
connectDevice()
Find new devices through mBroadCastReceiver
processMessage()
convert string to CBMessage
apply rules based on message type
display message
forward message
send()
convert CBMessage to bytes
for each connected device,
ConnectedThread.send()
ListenThread()
RequestThread (device)
processMessage (messageString)
connectedThread.write (bytes)
ListenThread
serverSocket.accept()
return socket and connected Bluetooth device
RequestThread
socket.connect()
return socket and connected Bluetooth device
ConnectedThread
socket.getInputStream()
socket.getOutputStream()
inStream.read()
parse input buffer and return message String
outStream.write()
write bytes to socket buffer
inStream.read()==exception
connection has failed
socket, BluetoothDevice
connectionLost(device)
ReconnectThread
run()
while (device.isNotConnected):
sleep(incrementalSleepTime)
connectDevice(device)
ReconnectThread(device)
ConnectionLost()
if (device==parent):
start ListenThread
sleep(1 minute)
if (device.isNotConnected):
start ReconnectThread
if (device==child):
attempts = 0
while (attempts<5) &
(device.isNotConnected()):
connectDevice(device)
sleep(10 sec)
if (device.isNotConnected):
for (routableDeviceOf(device):
start ReconnectThread

Figure 1: Diagram of source code interaction

THIS PAGE INTENTIONALLY LEFT BLANK

List of References

[1] C. Pellerin. (2013, Dec.). DARPA pioneers tactical mobile devices for soldiers. [Online]. Available: http://www.defense.gov/news/newsarticle.aspx?id=121320

[2] B. Coxworth, "U.S. Army trials tactical smartphones," *Gizmag*, Mar. 2011. [Online]. Available: http://www.gizmag.com/us-army-looks-into-tactical-smartphones/18152/

[3] K. McCaney, "Army gets tactical comm closer to the smartphone experience," *Defense Systems*, Apr. 2014. [Online]. Available: http://defensesystems.com/articles/2014/04/29/army-common-operating-environment.aspx

[4] W. Stallings, *Wireless Communications and Networks*, 2nd ed. Upper Saddle River, NJ: Pearson Education India, 2009, pp. 421–507.

[5] I. Poole. Bluetooth radio interface, modulation, and channels. [Online]. Available: http://www.radio-electronics.com/info/wireless/bluetooth/radio-interface-modulation.php

[6] "IEEE 802.15.1 Part 15.1: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Wireless Personal Area Networks (WPANs)," IEEE Standard 802.15, Tech. Rep., 2002.

[7] B. SIG, *Specification of the Bluetooth System, Version 4.2*, Bluetooth SIG, Dec. 02 2014.

[8] J. Bray, "Masters and Slaves: Roles in a Bluetooth Piconet," *InformIT*, May 2011. [Online]. Available: http://www.informit.com/articles/article.aspx?p=21324&seqNum=4

[9] *Bluetooth Specification: RFCOMM with TS 07.10*, vol. 12. Bluetooth SIG, Nov. 2012.

[10] T. Simonite, "The latest chat app for iPhone needs no internet connection," *MIT Technology Review*, Mar. 2014. [Online]. Available: http://www.technologyreview.com/news/525921/the-latest-chat-app-for-iphone-needs-no-internet-connection/

[11] *About Multipeer Connectivity*, Apple Inc., Sep. 2013. [Online]. Available: https://developer.apple.com/library/prerelease/ios/documentation/MultipeerConnectivity/Reference/MultipeerConnectivityFramework/index.html

[12] *Open Garden FAQs*, Open Garden, Apr. 2015. [Online]. Available: https://opengarden.com/faq

[13] T. Grasty. (2014, Oct.). How messaging app FireChat broke through the Great 'Firewall' of China. [Online]. Available: http://www.pbs.org/idealab/2014/10/how-messaging-app-firechat-broke-through-the-great-firewall-of-china/

[14] C. Baraniuk, "FireChat warns Iraqis that messaging app won't protect privacy," *Wired*, Jun. 2014. [Online]. Available: http://www.wired.co.uk/news/archive/2014-06/25/firechat

[15] F. Lardinois, "Firechat brings its anonymous offline chat network to android," *TechCrunch*, Apr. 2014. [Online]. Available: http://techcrunch.com/2014/04/03/firechat-brings-its-offline-chat-network-to-android/

[16] *LTE Direct: The case for device-to-device proximate discovery*, Qualcomm Technologies, Inc., San Diego, CA, 2013. [Online]. Available: https://www.qualcomm.com/documents/lte-direct-overview

Initial Distribution List

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California